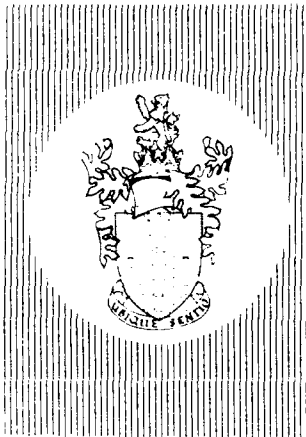


UNLIMITED

BR110569

②

Report No. 89004



Report No. 89004

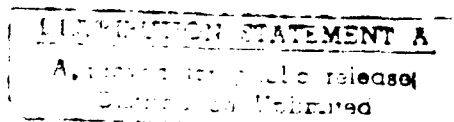
ROYAL SIGNALS AND RADAR ESTABLISHMENT,
MALVERN

AD-A210 838

DTIC
ELECTE
AUG 07 1989
S D cy D

PATTERN MATCHING IN ML:
A CASE STUDY IN REFINEMENT

Authors: R Macdonald, G P Randell, C T Sennett



PROCUREMENT EXECUTIVE, MINISTRY OF DEFENCE
RSRE
Malvern, Worcestershire.

May 1989

0044195

CONDITIONS OF RELEASE

BR-110583

U

COPYRIGHT (c)
1988
CONTROLLER
HMSO LONDON

Y

Reports quoted are not necessarily available to members of the public or to commercial organisations.

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Report 89004

Title: Pattern matching in ML: a case study in refinement

Authors: R Macdonald, G P Randell, C T Sennett

Date: May 1989

Summary

This report is a case study in data refinement, that is the process of taking a formal specification written in terms of abstract values and converting it into a concrete form suitable for implementation. The case study takes a non-trivial problem, namely pattern matching in the language ML, and presents the refinement process and proof obligations incurred concluding with an implementation in Algol68. The report concludes with a discussion of the strengths and weaknesses of the formal development process.

Copyright

©

Controller HMSO London

1989



| | |
|--------------------|--|
| Accession For | |
| NTIS CRA&I | <input checked="checked" type="checkbox"/> |
| DTIC TAB | <input type="checkbox"/> |
| Unannounced | <input type="checkbox"/> |
| Justification | |
| By | |
| Distribution / | |
| Availability Codes | |
| Avail and/or | Special |
| A-1 | |

Contents

| | |
|---|----|
| 1 Introduction | 1 |
| 2 Z specification of the problem | 4 |
| 2.1 Specification of ML values and types | 4 |
| 2.2 Specification of ML patterns and pattern matching | 7 |
| 2.3 The compatibility of types and patterns | 9 |
| 2.4 The compiling operations | 10 |
| 3 Formal implementation - the abstraction invariant | 13 |
| 4 The refined operations | 18 |
| 4.1 The initial operation | 18 |
| 4.2 The checking operation | 19 |
| 4.3 The final operation | 23 |
| 5 Abstract algorithm design | 24 |
| 5.1 The <i>coverage</i> function | 24 |
| 5.2 The <i>union</i> function | 26 |
| 6 Proof opportunities | 37 |
| 6.1 Proofs for the <i>coverage</i> function | 37 |
| 6.2 Proofs for the <i>union</i> function | 41 |
| 7 The implementation | 44 |
| 8 Conclusions | 51 |
| References | 54 |

1 Introduction

The trustworthiness of high integrity software is established by demonstrating that the software satisfies its specification. For certified software, the demonstration will be directed towards an independent evaluator who has to judge whether the software possesses the properties claimed for it. When the demonstration involves the use of formal methods the term refinement is used, standing for the process of producing the implementation in such a way that formal proof of satisfaction could be given. Consequently, refinement is an important technique for the production of software to the higher levels of assurance.

Refinement is usually associated with the final stages of producing an implementation. This arises from the need to use simple examples when explaining the concept, which are consequently rather close to the implementation language chosen. Most specifications are far removed from implementation languages, so the first steps of formal design will nearly always involve refining the specification. These steps may be carried out within the specification language itself and represent the major part of the creative design. Refinement is not an automatic method for the generation of an implementation from a specification and a key issue in its use is the extent to which it actually helps an implementor to produce trustworthy software. Another key issue is the extent to which the formality can help in demonstrating trustworthiness. In principle a formal proof forms a convincing argument but the details of the formalism may obscure the understanding of what is being proved. By hiding detail and automatically checking proof steps, tools could assist the process but with current technology the degree to which a proof helps rather than hinders the evaluation is an open question. The purpose of this report is to discuss these issues within the context of a realistic case study.

Section 2 presents a formal specification of the problem, which is concerned with pattern matching in the programming language ML. This is a problem which is interesting in its own right, but the formal refinement has a number of surprises which demonstrate the power of the method over the use of an intuitive approach.

Sections 3 to 5 deal with the formal development of the design specification. Section 3 deals with the choice of representation for the implementation. For the refinement method, the relationship between the entities in the abstract specification and those in the concrete implementation must be given and this is known as the abstraction invariant. Given this, the constraints on the implementation functions can actually be calculated, and this is done in section 4. With these in mind, section 5 deals with the design of the implementation algorithms at an abstract level which ensures that they are compatible with the specification. It is this aspect of abstract algorithm design where refinement has most to offer the implementor as opposed to the evaluator.

Section 6 covers the evaluator's requirements. It discusses the type of proof document which would illuminate the discussion rather than obscuring it and gives an example

of the way in which the salient features of the proof requirements could be displayed.

Section 7 provides an implementation of the problem in Algol68 which may be compared with the formal design specification.

Finally section 8 discusses the advantages and disadvantages of the formal approach and gives recommendations for current practice.

As some readers may not be familiar with ML the remainder of this introduction describes the pattern matching features of the language. ML was produced as a spin-off from the theorem proving system LCF [Gordon et al 1979, Paulson 1987]. In the latter, theorems, proof rules, tactics and so on are manipulated by means of an interactive *meta language*. As the ideas for this crystallised it was realised that the best meta language was in fact a programming language enriched with a polymorphic type discipline and exception handling. The language could be used quite generally in a similar manner to Lisp. The LCF idea produced a number of offspring, and varying dialects of ML began to appear. As a result, Milner convened the ML community with a view to developing a standard [Harper et al 1988]. The new language is larger and more powerful than its predecessors and is interesting in its own right, quite apart from its exciting use with theorem proving.

An attractive feature of the language is the use of *pattern matching* in the specification of functions. In ML, a function may be supplied as a series of clauses, each clause specifying how values of a certain structure are to be handled by the function as a whole. This is best illustrated by means of an example, for which the ML concepts of datatypes, patterns and pattern matching will be needed.

The ML *datatype* declaration is similar to a disjoint union and is a generalisation of the idea of an enumerated type in Pascal. The simplest datatype declarations take the form

datatype colour = red | blue | green

which declares *colour* as a type containing the three values *red*, *blue* and *green*. More usually, the datatype is constructed from previously defined types, as in the following declaration:

*datatype label = code of colour | number of int*int | name of string*

Thus a *label* value is constructed either from a *colour*, a pair of integers or a string. Instead of being constants as in the previous example, *code*, *number* and *name* are *constructor functions* which construct values having type *label*. So *code(red)*, *number(1, 2)* and *name "fred"* would all be values of type *label*.

An ML pattern is either a variable, an expression involving constants, constructor functions and variables or is a tuple of patterns. Thus, given variables *x*, *i*, *j*, then *x*,

number(i, j) and *code(red)* are all patterns. A pattern *matches* a set of values. A variable will match any value, but tuple patterns and patterns involving constructors only match values which have the corresponding structure. Thus *x* matches any value, *number(i, j)* matches any *label* value constructed from two integers and *code(red)* matches the single value it denotes.

A simple example of a function declared as a series of clauses is one which permutes the colours:

```
fun colour_perm red = blue
  | colour_perm blue = green
  | colour_perm green = red
```

This is both clearer and more concise than the corresponding expression involving conditionals. However, this expressiveness has been bought at the cost of some additional complexity in an ML compiler because it is necessary to check that the patterns supplied in the parameter position account for all the possible values to which the function might be applied. Thus *colour_perm* is a function from *colour* to *colour*, but if the last clause of its definition had been omitted it would only have been applicable to *red* and *blue* values. Consequently, it is a requirement on ML compilers that they should be able to check whether a set of patterns is *exhaustive*, that is, the patterns match every possible value belonging to a type. A related problem is that the addition of a clause to a function may be *redundant*, that is, it may not increase the number of values already matched. In this case the clause is superfluous and the programmer has made a mistake.

The two tests required are intuitively obvious in the simple case above, but when constructor functions and tuple values are considered, intuition can be misleading. In this case the formal specification is particularly helpful and it is presented here using the specification language Z [Sufrin 1983, Hayes 1987, Spivey 1988].

Notation: Sections 2 to 6 of this report are Z documents each of which has been mechanically type-checked. The conclusion of each document is marked by a "keeps" statement which lists the identifiers exported by the document. A document is imported into another one by including the document name, which is distinguished by having a box drawn round it.

2 Z specification of the problem

2.1 Specification of ML values and types

The specification is defined in terms of the operations a compiler must perform to carry out the test. A single pass of compilation is envisaged, so three operations are defined, an initial operation used to specify the compiler's action on encountering the first pattern of the clausal function, a testing operation to specify the action required on encountering the subsequent patterns and a final operation to test whether the set of patterns is exhaustive. The parameters for the second operation are the new pattern just compiled and the set of values matched by the patterns obtained from the other clauses of the function (the other patterns in the *match*, in ML terms). The check to be made is whether the set of values matched increased as a result of adding the latest pattern, in which case the match is not redundant. For the third operation, the parameter is the set of values accounted for by all the patterns in the match, and the test is whether this is all the values belonging to the type, in which case the set of patterns is exhaustive.

The problem is defined in terms of ML values so it is necessary to build a Z representation of them. In this a number of significant simplifications may be made. First of all, the treatment of the special constants (denotations for integers, reals and strings) is the same for each type of constant, so they are all represented together by the Z given set *SCONST*. Secondly, function values may only be matched by variables, which by definition are exhaustive. As the test is trivial in this case, function values are not considered further (they may be considered as being members of *SCONST*). Thirdly, the compiler's representation of constructors is irrelevant to the specification of the problem, so the constructors are simply introduced as a given set *CON*. Fourthly, the ML record type will simply be treated as though it were the corresponding tuple. Finally, polytypes will be ignored. A polymorphic match will be exhaustive or redundant at each and every instance of its type, so the problem is independent of the polymorphism.

With these simplifications, an ML value is either the special ML void or unit value or is drawn from the set of special constants or is a construction or a tuple of values. For the Z representation, a labelled, disjoint union will be used, but as the definition is recursive, *Value* is introduced as a given set. Thus values will be built up from the following given sets:

[*Value*, *SCONST*, *CON*]

An exhaustive set of patterns is one which matches every value of a given type and so the next step is to define what is meant by a type. The types are introduced as disjoint subsets of the values, with the actual type structure being given by constraints to be added later:

$$\begin{array}{|l}
Type : \mathbb{P} \mathbb{P}_1 \text{ Value} \\
\hline
\cup Type = Value \\
\forall T_1, T_2 : Type \mid T_1 \neq T_2 \cdot T_1 \cap T_2 = \emptyset
\end{array}$$

A constructed value is built from a constructor and a value for the parameter. The parameter value must be drawn from a specific type which was associated with the constructor in the datatype declaration. This association can be represented by the following (unspecified) function:

$$type_of_con : CON \rightarrow Type$$

Constructed values may now be described in terms of the following schema:

$$\begin{array}{|l}
\text{Cons_val} \text{ } \overline{\hspace{1cm}} \\
con : CON; val : Value \\
\hline
val \in type_of_con(con)
\end{array}$$

A constructor which is a constant will be treated as having the unit value as a parameter.

Tuples will simply be represented as sequences of two or more values. Tuples of patterns will also be required so it is helpful to make the generic definition:

$$\begin{array}{|l}
[T] \text{ } \overline{\hspace{1cm}} \\
tuple\ T == \{s : seq\ T \mid \#s > 1\}
\end{array}$$

With these definitions, the structure of values may now be represented by

$$Value ::= unitv \mid sconstv \ll SCONST \gg \mid consv \ll Cons_val \gg \mid tupv \ll tuple\ Value \gg$$

Given this simplified model of the values, the type structure is determined only by the constructors and tuples. The contribution a constructor makes to a type is represented by the set of constructors making up the datatype declaration. This is determined for each constructor and available to the compiler, so it is possible to presume the existence of the function below:

$$\begin{array}{|l}
datatype : CON \rightarrow \mathbb{P} CON \\
\hline
\forall cs_1, cs_2 : ran\ datatype \cdot cs_1 = cs_2 \vee cs_1 \cap cs_2 = \emptyset
\end{array}$$

The constraint in this partial specification ensures that the range of *datatype* partitions the set of constructors.

The values making up a type all possess the same structure determined by whether the values are tuples or constructed. A relation *same_type* on the values will be defined which specifies this property. The definition will be given by cases according to the structure of *Value*, and as it is recursive, the signature is given first:

$$_same_type_ : Value \leftrightarrow Value$$

For the primitive values, *unitv* forms a type on its own and all the special constants are considered to form one type. This is expressed by the schema:

| |
|--|
| <i>Primitive</i> |
| $v, w : Value$ |
| $v = unitv \wedge w = unitv$ \vee $v \in ran\ sconstv \wedge w \in ran\ sconstv$ |

Two constructed values have the same type if the constructors are drawn from one constructor set in the range of *datatype*:

| |
|--|
| <i>Constructions</i> |
| $v, w : Value$ |
| $v \in ran\ consv \wedge w \in ran\ consv$ $datatype\ (consv^{-1}\ v).con = datatype\ (consv^{-1}\ w).con$ |

As a consequence of this it is possible to deduce the set of constructors associated with a constructed type:

$$Constructors == \lambda\ ty : Type \mid ty \subseteq ran\ consv \bullet \{v : ty \bullet (consv^{-1}\ v).con\}$$

Finally, tuple values have the same type if they are the same length and the corresponding values have the same type:

| |
|--|
| <i>Tuples</i> |
| $v, w : Value$ |
| $v \in ran\ tupv \wedge w \in ran\ tupv$ $\#vs = \#ws$ $\forall\ i : 1 \dots \#vs \bullet (vs\ i)same_type(ws\ i)$ <p>where</p> $vs == tupv^{-1}\ v$ $ws == tupv^{-1}\ w$ |

As a tuple type is made up of values of the same tuple length it is possible to define a *Size* function to deliver it:

$$\begin{aligned} \text{Size} &== \lambda \text{ty} : \text{Type} \\ &\quad | \text{ty} \subseteq \text{ran tupv} \\ &\quad \bullet \mu n : \mathbb{N} \mid (\forall v : \text{tupv}^{-1}[\text{ty}] \bullet n = \#v) \bullet n \end{aligned}$$

The definition of *same_type* may now be completed as

$$\forall v, w : \text{Value} \bullet v \text{ same_type } w \Leftrightarrow \text{Primitive} \vee \text{Constructions} \vee \text{Tuples}$$

The set of types is simply the set of equivalence classes of *same_type*:

$$\forall \text{ty} : \text{Type} \bullet \forall v_1, v_2 : \text{ty} \bullet v_1 \text{ same_type } v_2$$

Consequently, it is possible to deduce the type of a value:

$$\text{type_of} == \lambda v : \text{Value} \bullet \{w : \text{Value} \mid w \text{ same_type } v\}$$

2.2 Specification of ML patterns and pattern matching

Patterns are formed in a similar way to values, except that one of the primitive patterns is a variable which matches any value. As with values, simplifications will be made: the ML layered pattern feature will be ignored and the wild card treated as a variable. For this problem it is not necessary to know the representation of variables, so they are introduced as a given set along with *Pattern* itself, introduced for the purposes of the recursive definition:

[*Pattern*, *Variable*]

Constructed patterns may be formed using the following schema:

$$\boxed{\begin{array}{l} \text{Cons_patt} \text{ —————} \\ \text{con} : \text{CON}; \text{patt} : \text{Pattern} \end{array}}$$

Using this, the structure of patterns is given by

$$\begin{aligned} \text{Pattern} ::= & \text{unitp} \mid \text{sconstp} \ll \text{SCONST} \gg \mid \text{var} \ll \text{Variable} \gg \mid \text{consp} \ll \text{Cons_patt} \gg \\ & \mid \text{tupp} \ll \text{tuple Pattern} \gg \end{aligned}$$

The definition of pattern matching is given in terms of a relation, *matches*, specifying which values match a given pattern:

$$_ \text{ matches } _ : \text{Pattern} \leftrightarrow \text{Value}$$

This will be defined recursively over the structure of patterns and values. For the

special constants a simplification will be made. The set of special constants is infinite and can never be matched by a finite set of patterns unless it contains a variable. For this case, it seems excessive to keep track of the special constants in a set of patterns simply in order to check for redundancy. Consequently the specification is relaxed to omit the redundancy check in this case. This will be done by allowing special constants to be matched by variables only. (Note that this is not necessary to our approach, but apart from being a sensible relaxation it simplifies the presentation.) For the other primitive patterns, the unit pattern matches the unit value and a variable matches any value. This is expressed by the following schema:

| |
|--|
| <i>MPrimitive</i> |
| $p : \text{Pattern}; v : \text{Value}$ |
| $p = \text{unit} \wedge v = \text{unit}$ |
| \vee |
| $p \in \text{ran var}$ |

A constructed pattern is matched by a constructed value if the constructors are the same and the parameters match:

| |
|---|
| <i>MConstructions</i> |
| $p : \text{Pattern}; v : \text{Value}$ |
| $p \in \text{ran consp} \wedge v \in \text{ran consv}$ |
| $p\text{con.con} = v\text{con.con} \wedge p\text{con.pat matches } v\text{con.val}$ |
| where |
| $p\text{con} == \text{consp}^{-1} p$ |
| $v\text{con} == \text{consv}^{-1} v$ |

Similarly, a tuple pattern matches a tuple value if the two tuples are the same size and corresponding elements match.

| |
|---|
| <i>MTuples</i> |
| $p : \text{Pattern}; v : \text{Value}$ |
| $p \in \text{ran tupp} \wedge v \in \text{ran tupv}$ |
| $\#tp = \#tv$ |
| $\forall i : 1 \dots \#tp \bullet (tp\ i) \text{ matches } (tv\ i)$ |
| where |
| $tp == \text{tupp}^{-1} p$ |
| $tv == \text{tupv}^{-1} v$ |

This gives, for the definition of *matches*:

$$\forall p : \text{Pattern}; v : \text{Value} \bullet p \text{ matches } v \Leftrightarrow M\text{Primitive} \vee M\text{Constructions} \vee M\text{Tuples}$$

The value coverage of a set of patterns is the set of values which may be matched to the patterns. As a variable matches any value it is necessary to restrict the set of values to one type so the compiling operations are specified in terms of a *valcover* function as follows:

$$\text{valcover} == \lambda \text{ patt} : \text{Pattern}; \text{ty} : \text{Type} \bullet \{v : \text{ty} \mid \text{patt matches } v\}$$

For a set of patterns *patts* of type *ty* to be exhaustive

$$\bigcup \{p : \text{patts} \bullet \text{valcover}(p, \text{ty})\} = \text{ty}$$

2.3 The compatibility of types and patterns

The implementation of the compiling operations is considerably simplified if it is possible to make use of the fact that the patterns are well-typed. This constraint may be expressed using a relation, similar to *same_type*, which specifies which patterns are compatible with a type. This relation is defined recursively in the usual way as follows:

$$\text{pattern_compatible_} : \text{Pattern} \leftrightarrow \text{Type}$$

For the primitive patterns, the unit pattern is compatible with the unit type, the special constant patterns are compatible with the special constant type and a variable is compatible with any type:

| |
|--|
| $PC\text{Primitive}$ |
| $p : \text{Pattern}; \text{type} : \text{Type}$ |
| $p = \text{unit} \wedge \text{type} = \{\text{unit}\}$ |
| \vee |
| $p \in \text{ran } \text{sconst} \wedge \text{type} = \text{ran } \text{sconst}$ |
| \vee |
| $p \in \text{ran } \text{var}$ |

For constructed patterns, the constructor used must be present in the set used to construct the values of the datatype, and the parameter must be compatible with the parameter type of the constructor:

| |
|---|
| <i>PCConstructions</i> |
| $p : \text{Pattern}; \text{type} : \text{Type}$ |
| $p \in \text{ran consp} \wedge \text{type} \subseteq \text{ran consv}$ $\text{con} \in \{\text{Cons_val} \mid \text{consv}(\theta \text{Cons_val}) \in \text{type} \bullet \text{con}\}$ $\text{patt pattern_compatible type_of_con}(\text{con})$ <i>where</i> Cons_patt $\text{consp}(\theta \text{Cons_patt}) = p$ |

A tuple pattern is compatible if the size of the tuple is the same as the tuple size of the type, and each element of the pattern is compatible with the types formed from the elements of the tuple type:

| |
|---|
| <i>PCTuples</i> |
| $p : \text{Pattern}; \text{type} : \text{Type}$ |
| $p \in \text{ran tupp} \wedge \text{type} \subseteq \text{ran tupv}$ $\#tp = \text{Size type}$ $\forall i : 1 \dots \#tp$ $\bullet (\text{tp } i) \text{ pattern_compatible } \{tv : \text{tupv}^{-1}[\text{type}] \bullet tv\ i\}$ <i>where</i> $tp == \text{tupp}^{-1} p$ |

$\forall p : \text{Pattern}; \text{type} : \text{Type}$
 $\bullet p \text{ pattern_compatible type} \Leftrightarrow \text{PCPrimitive} \vee \text{PCConstructions} \vee \text{PCTuples}$

For convenience, the predicate is defined as a schema:

| |
|---|
| <i>Pattern_Compatible</i> |
| $p : \text{Pattern}; \text{type} : \text{Type}$ |
| $p \text{ pattern_compatible type}$ |

2.4 The compiling operations

The initial operation generates the set of values from the first pattern to be compiled:

| |
|---|
| <i>Init_op</i> |
| $vals! : \mathbb{P} \text{ Value}; \text{ par?} : \text{ Pattern}; \text{ type} : \text{ Type}$ |
| $\text{par? pattern_compatible type} \wedge vals! = \text{valcover}(\text{par?}, \text{type})$ |

The type of the pattern being compiled is given by *type*, and as we are not concerned with type checking aspects of the compiler, this is treated as a constant throughout all the pattern checking operations.

It is convenient to express the result of the checking operation in terms of a new Z datatype:

Result ::= *OK* | *INCOMPLETE* | *REDUNDANT*

The checking operation is specified in terms of the set of values covered by the patterns compiled so far and the effect of adding one more pattern.

| |
|---|
| <i>Check_op</i> |
| $vals?, vals! : \mathbb{P} \text{ Value}$ |
| $\text{par?} : \text{ Pattern}$ |
| $r! : \text{ Result}$ |
| $\text{type} : \text{ Type}$ |
| $\text{par? pattern_compatible type} \wedge vals? \subseteq \text{type}$ |
| $vals! = \text{valcover}(\text{par?}, \text{type}) \cup vals?$ |
| $\text{valcover}(\text{par?}, \text{type}) \neq \{\} \wedge vals! = vals? \wedge r! = \text{REDUNDANT}$ |
| \vee |
| $(\text{valcover}(\text{par?}, \text{type}) = \{\} \vee vals! \neq vals?) \wedge r! = \text{OK}$ |

In this schema, *vals?* represents the set of values accounted for by the patterns compiled so far and *vals!* the result of adding the new pattern *par?*. The result of the operation is left in *r!*. The predicate $\text{valcover}(\text{par?}, \text{type}) \neq \{\}$ eliminates reporting a redundancy when the patterns include special constants.

The pre-condition of this operation is easily simplified to

$\text{par? pattern_compatible type} \wedge vals? \subseteq \text{type}$

If this is so, then from the definition of *valcover* it follows that *vals!* is also a subset of *type*, and for both operations. This is essential as the placing of the compiling operation with respect to the syntax of ML ensures that the output of *Init_op* will form the input to *Check_op* as does the output of *Check_op* itself. Strictly speaking, *type* is redundant in *Check_op* as it could be deduced from *vals?*. (A function *type_of*, which

gives the type of a value, has already been defined.) However, leaving the type in in this way leads to a clearer specification.

The final operation checks whether the set of patterns is exhaustive and is simply given by:

| |
|---|
| <i>Final_op</i> |
| $vals? : \mathbb{P} \text{ Value}; \text{ type} : \text{Type}; r! : \text{Result}$ |
| $vals? = \text{type} \wedge r! = \text{OK} \vee vals? \neq \text{type} \wedge r! = \text{INCOMPLETE}$ |

As with the other operations, the ML syntax determines when this operation is called: the input is provided either by *Init_op* for a one-pattern match, or by the last call of *Check_op* in a multi-pattern match.

This completes the specification of the problem which should be checked for validity, that is, that it accurately captures the essence of what has been informally specified in the ML definition.

Z_match_spec keeps *Value*, *Cons_val*, *datatype*, *unitv*, *sconstv*, *consv*, *tupv*, *Cons_patt*, *Constructors*, *Size*, *Type*, *SCONST*, *CON*, *type_of_con*, *tuple*, *Pattern*, *unitp*, *var*, *sconstp*, *consp*, *tupp*, *matches*, *MPrimitive*, *MConstructions*, *MTuples*, *PCPrimitive*, *PCConstrutions*, *PCTuples*, *pattern_compatible*, *Pattern_Compatible*, *Result*, *OK*, *INCOMPLETE*, *REDUNDANT*, *valcover*, *Init_op*, *Check_op*, *Final_op*

3 Formal implementation - the abstraction invariant

Z_match_spec :Module

The specification has been written in a form which matches as closely as possible the informal specification given in the language definition. As such it is defined in terms of possibly infinite sets of values and it is not possible to implement a test such as *vals? = type* in *Final_op* directly. Instead a *coverage* measure is used to keep track of how completely a set of patterns accounts for the values in a type. Data refinement is used to specify the operations on coverages which correspond to the operations in the specification. The key step in data refinement is to define the abstraction invariant which specifies the set of values which correspond to a given coverage.

The first question to be settled in carrying out the data refinement is the form for the coverage measure. The form chosen must be easily implementable and the check not excessively time-consuming. The essentials of the problem are that for constructed patterns every constructor must be accounted for and for tuple patterns the individual elements must be complete. This latter property is surprisingly hard to formalise, so it is useful to have a few test cases to clarify what is actually required. Using the datatype definitions given previously, consider the case of a 3-tuple of colours. The following sequence of patterns is complete and not redundant in the sense that each succeeding pattern matches more values and the complete set of values is only accounted for with the final pattern.

| Number of extra values matched | |
|--------------------------------|---|
| <i>(red, blue, green)</i> | 1 |
| <i>(x, blue, green)</i> | 2 |
| <i>(red, x, green)</i> | 2 |
| <i>(red, blue, x)</i> | 2 |
| <i>(red, x, y)</i> | 4 |
| <i>(blue, green, red)</i> | 1 |
| <i>(blue, x, y)</i> | 7 |
| <i>(x, green, green)</i> | 1 |
| <i>(x, red, green)</i> | 1 |
| <i>(x, y, z)</i> | 6 |

It is worth while checking this table to convince yourself that if the concept of exhaustive patterns is intuitive, the actual check to apply in the tuple case is not. In order to handle tuples generally, the coverage measure adopted will involve a function from the coverage provided by the first element of the tuple to the coverage provided by the remainder of the tuple, rather than having a tuple of coverages. This will become clearer as the formalism is developed.

The other complication in the problem is that it is necessary to account for constructor functions as well as constants in the patterns. Thus for a pair of *labels* one has the following sequence of exhaustive but not redundant patterns:

```

(code(red), x)
(x, number(0, y))
(code(x), y)
(x, name(y))
(x, y)

```

It is possible to treat constructed patterns as though they were 2-tuples, using a function in exactly the same way as for tuples. This approach has not been taken in the implementation presented here, partly for reasons of efficiency and partly for ease of understanding. Instead, a function from constructors to coverages is used.

Special constants are treated by having an *incomplete* coverage measure, which represents an empty set of values, corresponding to the fact that, in the specification, a special constant pattern matches no value. Treating the special constants accurately would require keeping a measure dependent on the number of constants already accounted for.

With this motivation the Z datatype defining the coverage measure may be written down:

$$\text{Cover} ::= \text{complete} \mid \text{incomplete} \\ \mid \text{construct} \ll \text{CON} \Rightarrow \text{Cover} \gg \mid \text{pair} \ll \mathbb{F} (\text{Cover} \times \text{Cover}) \gg$$

Thus the coverage measure chosen is a tree in which the leaves correspond to full or no coverage and the nodes of the tree correspond to the constructor and tuple structure of the type. The parameter of *pair* is given as a set of pairs rather than a function as this seems to make the explanation easier. Finite sets are used to ensure that the datatype is satisfiable. Before going on to define the abstraction function, it may be mentioned that the sequence of patterns compiled also forms a coverage measure. However, it is worth having a separate coverage datatype in order to optimise the operations required.

The formal definition of the abstraction invariant will be motivated by giving an example of the intended relation between coverages and patterns. The first two patterns from the sequence of *label* pairs above are to be represented by the following two coverages:

```

pair {construct {code  $\mapsto$  construct {red  $\mapsto$  complete}}  $\mapsto$  complete}
pair {complete  $\mapsto$  construct {number  $\mapsto$  pair {incomplete  $\mapsto$  complete}}}

```

The essence of the problem is to combine successive coverages into one in such a way as to end up with the *complete* cover when the set of patterns is exhaustive.

For the presentation of the abstraction invariant some functions for manipulating tuple types are necessary. From a tuple type one can form a type corresponding to the first element of the tuple and one corresponding to the remainder of the tuple, given by two functions *HD* and *TL* as follows:

$$\begin{aligned}
HD == & \lambda \text{ ty} : \text{Type} \\
& | \text{ ty} \in \text{ran tupv} \\
& \bullet \{ \text{tv} : \text{tuple Value} \mid \text{tv} \in \text{tupv}^{-1} [\text{ty}] \bullet \text{hd tv} \}
\end{aligned}$$

| |
|--|
| $TL : \text{Type} \rightarrow \text{Type}$ |
| $\forall \text{ ty}_1, \text{ ty}_2 : \text{Type}$ $ \text{ ty}_1 \in \text{ran tupv}$ $\bullet \text{ ty}_2 = TL \text{ ty}_1$ \Leftrightarrow $\text{Size ty}_1 = 2 \wedge \text{ ty}_2 = \{ \text{tv} : \text{tuple Value} \mid \text{tv} \in \text{tupv}^{-1} [\text{ty}_1] \bullet \text{tv}(2) \}$ \vee $\text{Size ty}_1 > 2 \wedge \text{ ty}_2 = \{ \text{tv} : \text{tuple Value} \mid \text{tv} \in \text{tupv}^{-1} [\text{ty}_1] \bullet \text{tupv}(\text{tl tv}) \}$ |

Note that these functions deliver a type rather than an arbitrary set of values because successive elements of values in a tuple type must have the same type.

The basic requirement for the abstraction is a function from a *Cover* value to a set of *Value*. This cannot be provided from the datatype chosen because the set of values from a *complete* cover will depend on the type. However, the actual check to be made is independent of the type in this case, so the specification contains redundant information as far as these particular checks are concerned. (This state of affairs is called *bias*.) It is still possible to carry out the refinement, but in order to do so, it is necessary to have a function from a *Cover* and a *Type* to a set of *Value*. A unique definition of this function will be provided, built up according to the structure of *Cover* and *Type* in the usual way.

$$Abs_fn : (\text{Cover} \times \text{Type}) \rightarrow \mathbb{P} \text{ Value}$$

For the primitive coverage elements we have the following

| |
|---|
| $AI\text{Primitive}$ |
| $c : \text{Cover}; \text{ type} : \text{Type}; \text{ vals} : \mathbb{P} \text{ Value}$ |
| $(c = \text{complete} \wedge \text{vals} = \text{type}) \vee (c = \text{incomplete} \wedge \text{vals} = \{\})$ |

For constructed coverages the cover represents any constructed value it is possible to form from a constructor in the domain of the constructor function combined with a value drawn from the set represented by the corresponding coverage element in the range of the function. This is expressed by the following schema:

| |
|--|
| $ \begin{array}{l} \text{AIConstructions} \\ \hline c : \text{Cover}; \text{type} : \text{Type}; \text{vals} : \mathbb{P} \text{ Value} \\ \hline c \in \text{ran construct} \wedge \text{type} \subseteq \text{ran consv} \\ \text{vals} = \{ \text{Cons_val}; F : \text{CON} \leftrightarrow \text{Cover} \\ \quad c = \text{construct } F \wedge \text{con} \in \text{dom } F \\ \quad \wedge \text{val} \in \text{Abs_fn}(F \text{ con}, \text{type_of_con}(\text{con})) \\ \quad \bullet \text{ consv } \theta \text{Cons_val} \\ \quad \} \end{array} $ |
|--|

For tuples we need a function from pairs of coverages to sets of values as follows:

| |
|--|
| $ \begin{array}{l} \text{Tuple_vals} : (\text{Cover} \times \text{Cover} \times \text{Type}) \leftrightarrow \mathbb{P} \text{ tuple Value} \\ \hline \forall c_1, c_2 : \text{Cover}; \text{type} : \text{Type}; \text{vals} : \mathbb{P} \text{ tuple Value} \\ \text{type} \subseteq \text{ran tupv} \\ \bullet \text{Tuple_vals}(c_1, c_2, \text{type}) = \text{vals} \\ \Leftrightarrow \\ \text{Size type} = 2 \wedge \text{vals} = \{ v_1 : \text{Abs_fn}(c_1, \text{HD type}); v_2 : \text{Abs_fn}(c_2, \text{TL type}) \\ \quad \bullet \langle v_1, v_2 \rangle \\ \quad \} \\ \vee \\ \text{Size type} > 2 \wedge \text{vals} = \{ v_1 : \text{Abs_fn}(c_1, \text{HD type}); pc_1, pc_2 : \text{Cover}; \\ \quad tv : \text{tuple Value} \\ \quad pc_1 \mapsto pc_2 \in \text{pair}^{-1} c_2 \\ \quad \wedge tv \in \text{Tuple_vals}(pc_1, pc_2, \text{TL type}) \\ \quad \bullet v_1 \text{ cons } tv \\ \quad \} \end{array} $ |
|--|

The set of values corresponding to a pair of covers is that set of tuple values formed by taking any element from the set corresponding to the first cover for the first element and combining it with any element from the set corresponding to the second. Thus if the first cover corresponds to n values and the second to m , the pair of covers corresponds to $n \times m$ tuple values. For longer tuples, the second cover value will itself be a pair giving rise to a set of tuple values whose size is one less than the original. Form a set by taking any element from the values corresponding to the first cover and add this to the front of any element in the tuples provided by the second cover.

With this auxiliary function the abstraction function for tuples may be defined as follows:

AITuples

$c : \text{Cover}; \text{type} : \text{Type}; \text{vals} : \mathbb{P} \text{ Value}$

$c \in \text{ran pair} \wedge \text{type} \subseteq \text{ran tupv}$

$\text{vals} = \bigcup \{c_1, c_2 : \text{Cover} \mid c_1 \mapsto c_2 \in \text{pair}^{-1} c \bullet \text{tupv} \llbracket \text{Tuple_vals}(c_1, c_2, \text{type}) \rrbracket \}$

The abstraction function itself is simply given by the constraint:

$\text{Abs_fn} = \lambda c : \text{Cover}; \text{type} : \text{Type}$

$\bullet \mu \text{vals} : \mathbb{P} \text{ Value} \mid \text{AIPrimitive} \vee \text{AIConstructions} \vee \text{AITuples} \bullet \text{vals}$

Finally, it is convenient to define the abstraction invariant as a schema:

AI

$c : \text{Cover}; \text{type} : \text{Type}; \text{vals} : \mathbb{P} \text{ Value}$

$\text{vals} = \text{Abs_fn}(c, \text{type})$

Z_match_AI keeps *Cover*, *complete*, *incomplete*, *construct*, *pair*,
HD, *TL*,
AI, *Abs_fn*, *AIPrimitive*, *AIConstructions*, *AITuples*

4 The refined operations

4.1 The initial operation

$Z_match_spec : Module$

$Z_match_AI : Module$

It is interesting to follow the technique given in Morgan [1988] in which the concrete operations corresponding to the abstract operations are actually calculated from the abstraction invariant. To summarise the notation, which will be slightly adapted from that used by Morgan, an operation is represented by its pre and post-conditions as below:

$$\Delta av [pre, post]$$

This represents an operation achieving a state of affairs specified by the predicate *post*. It must be given an initial state represented by *pre* and achieves it by altering the abstract variable *av*.

Given an abstraction invariant *AI*, involving the concrete variable *cv*, the corresponding concrete operation is simply given by the formula

$$\Delta cv [\exists av \bullet AI \wedge pre, \exists av \bullet AI \wedge post]$$

This technique will be applied first of all to the operation *Init_op*. In the refinement notation we can write:

$$Init_op \sqsubseteq \Delta vals [par\ pattern_compatible\ type, vals = valcover(par, type)]$$

This statement is an assertion that the operation on the right hand side of the \sqsubseteq symbol is an operation refinement of the Z schema operation *Init_op*. In the refinement notation, operations are expressed in terms of variables assumed to be declared within the current context of the operation. Variables in the pre-condition refer to values before the operation while variables in the post-condition refer to values after. Consequently, there is no need for the Z decorations of !, ? or ' and these are systematically dropped. The pre-condition for the operation has been derived by existentially quantifying over the output variables in the Z schema and simplifying.

For the data refinement, the concrete variable is the coverage, *c*, of type *Cover* while the abstract variable is the set of values *vals*, of type $\mathbb{P} Value$. The concrete operations will eventually prove to be independent of the type which may simply be discarded. Using the abstraction invariant and the data refinement symbol \sqsubseteq , our operation is refined as follows:

$$\begin{aligned} \sqsubseteq \Delta c [\exists vals : \mathbb{P} Value \bullet AI \wedge par\ pattern_compatible\ type, \\ \exists vals : \mathbb{P} Value \bullet AI \wedge vals = valcover(par, type)] \end{aligned}$$

This operation is guaranteed to be a correct data refinement of *Init_op*. Looking at the post-condition, it is clear that it is necessary to calculate a coverage from the input pattern. This coverage must be such that the application of the abstraction function gives the same set of values as that provided by *valcover* when applied to the input pattern. It is fairly obvious, once one has stood back from the trees of the formalism to view the wood of the problem, that the initial value of *c* is irrelevant. Consequently there is no problem incurred in weakening the pre-condition and simplifying the post-condition by substituting for *vals* as follows

$$\models \Delta c \text{ [par pattern_compatible type, Abs_fn}(c, \text{type}) = \text{valcover}(\text{par}, \text{type})]$$

The implementation problem therefore is to define a *coverage* function which relates a *Pattern* to a *Cover* in the manner required:

$$\text{coverage} : \text{Pattern} \rightarrow \text{Cover}$$

For this to be a correct implementation of the initial operation, the following theorem has to be proved:

$$\text{Pattern_Compatible}$$

⊢

$$\text{Abs_fn}(\text{coverage}(p), \text{type}) = \text{valcover}(p, \text{type})$$

The specification of *coverage* will be deferred to a later section because the other operations introduce further constraints on its definition.

4.2 The checking operation

Proceeding in the same way as before

$$\text{Check_op} \models \text{con } \text{vals}_0 \bullet \Delta \text{vals}, r \text{ [vals} = \text{vals}_0 \wedge \text{par pattern_compatible type} \\ \wedge \text{vals} \subseteq \text{type, Check_op}]$$

This step has introduced some more refinement notation. Where, as in this case, the post-condition is dependent on the values both before and after the operation, it is necessary to preserve the initial value using a *logical constant*, which is introduced with the reserved word *con*. By convention, initial values are indicated with a 0-subscript, so *vals*₀ corresponds to the input value *vals*? in the schema.

Note that in this specification the pre-condition does not record the fact that the input values are provided from the results of the initial operation or a previous use of the checking operation as the case may be. This will be introduced informally later. The data refinement for the checking operation can be written as:

$$\begin{aligned} \leq \text{con } \text{vals}_0, c_0 \bullet \Delta c, r \ [\exists \text{vals} : \mathbb{P} \text{Value} \\ \bullet \text{AI} \wedge \text{vals} = \text{vals}_0 \wedge c = c_0 \\ \wedge \text{par pattern_compatible type} \wedge \text{vals} \subseteq \text{type}, \\ \exists \text{vals} : \mathbb{P} \text{Value} \bullet \text{AI} \wedge \text{Check_op}] \end{aligned}$$

Because $\text{Abs_fn}(c, \text{type})$ is always a subset of type , the pre-condition may be replaced by $\text{vals}_0 = \text{Abs_fn}(c_0, \text{type}) \wedge \text{par pattern_compatible type}$.

The post-condition may be simplified, by eliminating vals and vals_0 , into a predicate described by the following schema:

| |
|--|
| <p><i>Check_op_1</i></p> <hr/> <p>$c, c_0 : \text{Cover}$ $\text{par} : \text{Pattern}$ $\text{type} : \text{Type}$ $r : \text{Result}$</p> <hr/> <p>$\text{par pattern_compatible type}$ $\text{Abs_fn}(c, \text{type}) = \text{valcover}(\text{par}, \text{type}) \cup \text{Abs_fn}(c_0, \text{type})$ $\text{valcover}(\text{par}, \text{type}) \neq \{\} \wedge \text{Abs_fn}(c, \text{type}) = \text{Abs_fn}(c_0, \text{type})$ $\wedge r = \text{REDUNDANT}$ $\vee (\text{valcover}(\text{par}, \text{type}) = \{\} \vee \text{Abs_fn}(c, \text{type}) \neq \text{Abs_fn}(c_0, \text{type}))$ $\wedge r = \text{OK}$</p> |
|--|

We already have the requirement to find a coverage function such that $\text{valcover}(\text{par}?, \text{type}) = \text{Abs_fn}(\text{coverage}(\text{par}?), \text{type})$, so it is tempting to define a union function between coverages which carries out the corresponding operation to forming a union of sets of values:

$$\text{union} : (\text{Cover} \times \text{Cover}) \rightarrow \text{Cover}$$

and refine to the operation

| |
|---|
| <p><i>Check_op_2</i></p> <hr/> <p>$c, c_0 : \text{Cover}$ $\text{par} : \text{Pattern}$ $\text{type} : \text{Type}$ $r : \text{Result}$</p> <hr/> <p>$c = \text{union}(\text{coverage}(\text{par}), c_0)$ $\text{coverage}(\text{par}) \neq \text{incomplete} \wedge c = c_0 \wedge r = \text{REDUNDANT}$ $\vee (\text{coverage}(\text{par}) = \text{incomplete} \vee c \neq c_0) \wedge r = \text{OK}$</p> |
|---|

For this to be so, the following theorem must be proved:

$$\begin{array}{l} \text{Check_op_2} \\ \vdash \\ \text{Check_op_1} \end{array}$$

This corresponds to strengthening the post-condition, namely, that the achievement of *Check_op_2* will entail the achievement of *Check_op_1*. Note that the type has now dropped out of the predicates.

For the proof of this theorem, it is necessary to show that the *union* function behaves like set union and that there is a unique representation of the empty set of values. It is difficult to define a function having the properties required without taking into account the fact that the coverages have all been derived from patterns of the same type. Consequently, extra constraints will be added to the specification which are satisfied by the pre-condition. These constraints will be defined in terms of a *cover_compatible* relation, and express the fact that the *union* function is only required to combine coverages of the same structure.

This is defined in a similar way to *pattern_compatible*:

$$\text{_cover_compatible_} : \text{Cover} \leftrightarrow \text{Type}$$

$$\begin{array}{l} \text{CCPrimitive} \text{ —————} \\ c : \text{Cover}; \text{type} : \text{Type} \\ \hline c = \text{complete} \vee c = \text{incomplete} \end{array}$$

$$\begin{array}{l} \text{CCConstructions} \text{ —————} \\ c : \text{Cover}; \text{type} : \text{Type} \\ \hline c \in \text{ran construct} \wedge \text{type} \subseteq \text{ran consv} \\ \text{dom } F \subseteq \{\text{Cons_val} \mid \text{consv}(\theta \text{Cons_val}) \in \text{type} \bullet \text{con}\} \\ \forall \text{con} : \text{dom } F \bullet (F \text{con}) \text{cover_compatible} (\text{type_of_con con}) \\ \text{where} \\ F == \text{construct}^{-1} c \end{array}$$

| |
|---|
| <i>CCTuples</i> |
| $c : \text{Cover}; \text{type} : \text{Type}$ |
| $c \in \text{ran pair} \wedge \text{type} \subseteq \text{ran tupv}$ $\forall c : \text{dom } F \bullet c \text{ cover_compatible HD type}$ $\forall c : \text{ran } F \bullet c \text{ cover_compatible TL type}$ <i>where</i> $F == \text{pair}^{-1} c$ |

$\forall c : \text{Cover}; \text{type} : \text{Type}$
 $\bullet c \text{ cover_compatible type} \Leftrightarrow \text{CCPrimitive} \vee \text{CCConstructions} \vee \text{CCTuples}$

In the refined operation, the compatibility of the coverage c with the type is guaranteed by the compatibility of the pattern par with the type while the fact that c_0 is compatible is guaranteed by the initial set of values in the abstract operation being generated by matching type-compatible patterns. This is all rather tedious to formalise and not very illuminating, so these theorems will not be stated. The theorems expressing the more interesting union and uniqueness properties may be broken down into sub-goals defined in terms of the following schema, which gathers together the parameters of *union* and its result, and has the type as a parameter:

| |
|--|
| <i>Union</i> |
| $c_1, c_2, c : \text{Cover}$ $\text{type} : \text{Type}$ |
| $c = \text{union}(c_1, c_2)$ $c_1 \text{ cover_compatible type}$ $c_2 \text{ cover_compatible type}$ |

The first goal is related to the first constraint of *Check_op_2*:

Union
 \vdash
 $\text{Abs_fn}(c, \text{type}) = \text{Abs_fn}(c_1, \text{type}) \cup \text{Abs_fn}(c_2, \text{type})$

The remaining goals are related to the second constraint:

Pattern_Compatible
 \vdash
 $\text{coverage}(p) \neq \text{incomplete} \Rightarrow \text{valcover}(p, \text{type}) \neq \{\}$

Union
 \vdash
 $c = c_2 \Rightarrow \text{Abs_fn}(c, \text{type}) = \text{Abs_fn}(c_2, \text{type})$

Union

\vdash

$$c \neq c_2 \Rightarrow \text{Abs_fn}(c, \text{type}) \neq \text{Abs_fn}(c_2, \text{type})$$

The first of these goals requires the *coverage* function to deliver the *incomplete* value whenever the value coverage is empty. The second is trivially true and the third requires the result of the *union* function to change whenever the set of values covered changes. Further constraints on *union* will emerge with the definition of the final operation.

4.3 The final operation

This performs the final check for whether the set of patterns is exhaustive or not. The operation is simply refined as follows:

$$\begin{aligned} \text{Final_op} &\models \Delta r [\text{true}, \text{vals} = \text{type} \wedge r = \text{OK} \vee \text{vals} \neq \text{type} \wedge r = \text{INCOMPLETE}] \\ &\leq \Delta r [\text{true}, c = \text{complete} \wedge r = \text{OK} \vee c \neq \text{complete} \wedge r = \text{INCOMPLETE}] \end{aligned}$$

Here the data refinement and simplification of the pre and post conditions have been carried out in one step. The input to the operation is simply the result of *union*, or, in the case of a single clause in the function definition, the result of *coverage*. Accordingly, there are further constraints to satisfy. These are:

Union

\vdash

$$c = \text{complete} \Rightarrow \text{Abs_fn}(c, \text{type}) = \text{type}$$

Union

\vdash

$$c \neq \text{complete} \Rightarrow \text{Abs_fn}(c, \text{type}) \neq \text{type}$$

$p : \text{Pattern}; \text{type} : \text{Type}; c : \text{Cover}$

\vdash

$$c = \text{coverage}(p) = \text{complete} \Rightarrow \text{Abs_fn}(c, \text{type}) = \text{type}$$

$p : \text{Pattern}; \text{type} : \text{Type}; c : \text{Cover}$

\vdash

$$c = \text{coverage}(p) \neq \text{complete} \Rightarrow \text{Abs_fn}(c, \text{type}) \neq \text{type}$$

The first and third of these goals follow immediately from the definition of the abstraction function, while the second and fourth again require a unique representation of completeness.

Z_match_ops keeps coverage, union, Union,
cover_compatible, CCPrimitive, CCConstructions, CCTuples

5 Abstract algorithm design

$Z_match_spec : Module$

$Z_match_AI : Module$

$Z_match_ops : Module$

5.1 The coverage function

We have to provide a constructive definition of the function which satisfies the following theorems:

Pattern_Compatible

\vdash

$Abs_fn(coverage(p), type) = valcover(p, type)$

Pattern_Compatible

\vdash

$coverage(p) \neq incomplete \Rightarrow valcover(p, type) \neq \{\}$

Pattern_Compatible; c : Cover

\vdash

$c = coverage(p) = complete \Rightarrow Abs_fn(c, type) = type$

Pattern_Compatible; c : Cover

\vdash

$c = coverage(p) \neq complete \Rightarrow Abs_fn(c, type) \neq type$

The first of these gives the basic property required, the second the property that the empty value coverage is uniquely represented by the *incomplete* cover and the last two that the *complete* cover uniquely represents the exhaustive set of patterns. Note that the third of these theorems follows immediately from the definition of *Abs_fn*. With these theorems in mind, the definition of *coverage* may be written down by cases on the structure of *Pattern*, following the definition of *matches*:

For the primitive patterns of variables and special constants we have

CPrimitive

$p : Pattern; c : Cover$

$p = unitp \wedge c = complete$

\vee

$p \in ran\ var \wedge c = complete$

\vee

$p \in ran\ sconstp \wedge c = incomplete$

For constructed patterns a *construct* coverage will normally be produced in which the first element corresponds to the constructor and the second to the coverage provided by the parameter. To satisfy the second theorem above, it is necessary to test for an *incomplete* parameter coverage while to satisfy the fourth theorem the *complete* cover must be returned when the set of patterns is exhaustive. For a datatype containing only one constructor this is the case when the parameter provides a complete coverage and this case must be tested.

| |
|--|
| <i>CConstructions</i> |
| $p : \text{Pattern}; c : \text{Cover}$ |
| $p \in \text{ran consp}$ $\text{datatype con} = \{\text{con}\} \wedge \text{parcover} = \text{complete} \wedge c = \text{complete}$ \vee $\text{parcover} = \text{incomplete} \wedge c = \text{incomplete}$ \vee $\neg(\text{parcover} = \text{incomplete} \vee \text{datatype con} = \{\text{con}\} \wedge \text{parcover} = \text{complete}) \wedge$ $c = \text{construct } \{\text{con} \mapsto \text{parcover}\}$ <i>where</i> $\text{con} == (\text{consp}^{-1} p).\text{con}$ $\text{parcover} == \text{coverage } (\text{consp}^{-1} p).\text{patt}$ |

For tuples, it is also necessary to test for *complete* and *incomplete* partial results, which is done using the *CPair* function as follows:

```

CPair == λ c1, c2 : Cover
  • μ c : Cover
    | c1 = complete ∧ c2 = complete ∧ c = complete
    ∨
    (c1 = incomplete ∨ c2 = incomplete) ∧ c = incomplete
    ∨
    ¬(c1 = incomplete ∨ c2 = incomplete)
    ∧
    ¬(c1 = complete ∧ c2 = complete)
    ∧ c = pair {c1 ↦ c2}
  • c

```

Note that we are now getting to the stage where the *if then else* notation of the implementation language would be more natural and more compact.

With this function the coverage for tuples is given by

| |
|---|
| $CTuples$ $p : Pattern; c : Cover$ |
| $p \in ran\ tupp$ $\#tp = 2 \wedge c = CPair(coverage(tp\ 1), coverage(tp\ 2))$ \vee $\#tp > 2 \wedge c = CPair(coverage(tp\ 1), coverage(tupp(1l\ tp)))$ <i>where</i> $tp == tupp^{-1} p$ |

The coverage function is given by the constraint:

$$coverage = \lambda p : Pattern \cdot \mu c : Cover \mid CPrimitive \vee CConstructions \vee CTuples \cdot c$$

Note that this function is total as the disjunction covers all possible *Pattern* constructors.

5.2 The *union* function

As with *coverage*, the *union* function must satisfy the following theorems:

Union

$$\vdash Abs_fn(c, type) = Abs_fn(c_1, type) \cup Abs_fn(c_2, type)$$

Union

$$\vdash c = c_2 \Rightarrow Abs_fn(c, type) = Abs_fn(c_2, type)$$

Union

$$\vdash c \neq c_2 \Rightarrow Abs_fn(c, type) \neq Abs_fn(c_2, type)$$

Union

$$\vdash c = complete \Rightarrow Abs_fn(c, type) = type$$

Union

$$\vdash c \neq complete \Rightarrow Abs_fn(c, type) \neq type$$

The principal objective is to make the *union* function correspond to the operation of uniting sets of values. The additional constraints are that the coverage must only change when the underlying sets of values change and that it is necessary to have a unique measure for the exhaustive set of patterns.

For the abstract algorithm design, consider first the case of a set of constructed patterns. These are represented by a *construct* coverage which measures the coverage of values associated with each of the constructors in the datatype. So the coverage will be represented by some function, F , of the form $F = \{c_i \mapsto P_i\}$, where the c_i are the constructors of a datatype and the P_i represent sets of parameter values covered so far. Adding a new pattern will give rise to an extra coverage represented in the same way by the maplet $c_j \mapsto P$. The new coverage, F' , will depend on whether c_j is a member of the domain of F or not. If it is, the coverage provided by the parameter of the new pattern is united with the coverage provided by the parameters of the patterns already processed which use that constructor. This is expressed formally as $F' = F \oplus \{c_j \mapsto P_j \cup P\}$. When a new pattern introduces a constructor for the first time, the coverage is simply added to those already there. In this case, $F' = F \cup \{c_j \mapsto P\}$.

Calculating the united coverage in this way will cause F' to differ from F exactly when a new constructor is added to the coverage or when a parameter coverage, *parc*, changes. This satisfies the requirement of the third goal for this case. For the fifth goal, a test for completeness is required, expressed as follows:

$$\begin{array}{|l} \text{Constructor_complete} \\ F' : \text{CON} \mapsto \text{Cover} \\ \hline \exists cs : \text{ran datatype} \cdot F' = \{c : cs \cdot c \mapsto \text{complete}\} \end{array}$$

Using this schema and generalising to cover the case of merging sets of patterns, the *union* constructor case is defined as follows:

$$\begin{array}{|l} \text{UConstructions} \\ c_1, c_2, c : \text{Cover} \\ \hline c_1 \in \text{ran construct} \wedge c_2 \in \text{ran construct} \\ \text{Constructor_complete} \wedge c = \text{complete} \\ \vee \\ \neg \text{Constructor_complete} \wedge c = \text{construct } F' \\ \text{where} \\ F == \text{construct}^{-1} c_1 \\ f == \text{construct}^{-1} c_2 \\ f_1 == \text{dom } f \uplus F \\ f_2 == \text{dom } F \uplus f \\ f_3 == \{c : \text{dom } F \cap \text{dom } f \cdot c \mapsto \text{union}(F c, f c)\} \\ F' == (f_1 \cup f_2 \cup f_3) \end{array}$$

This is fairly straightforward, but the same technique may be used to deal with tuple

patterns, which are represented by *pair* coverages. In this case, one coverage stands for the set of values covered in the first element of the tuple, which is bound to the set of values covered by the rest of the tuple in a similar way to that in which the parameter of a constructed pattern is bound to its constructor. Consequently a pair coverage stands for a set of values which may be represented in the form $F = \{A_i \mapsto B_i\}$, where the A s and B s now stand for sets of values. Each element of the set F stands for a set of tuple values formed by taking one element out of an A and combining it with any element out of the corresponding B . A new pattern gives rise to a set of values covered of the form $\alpha \mapsto \beta$. For a given element of F , say $A \mapsto B$, the standard rules for taking unions of Cartesian products should give rise to the following extra elements in F' :

$$\begin{aligned} A \setminus \alpha &\mapsto B \\ A \cap \alpha &\mapsto B \cup \beta \\ \alpha \setminus A &\mapsto \beta \end{aligned}$$

Applying this procedure to every element in F gives the new coverage, F' . If any of the sets are empty, this element represents no values and may be discarded. To meet the other goals it is important that F' should differ from F exactly when new values have been added to the set of values covered. This objective is attained by keeping the A_i disjoint: an alteration to one of them cannot then produce a value which is already accounted for by the other members of F . If this is the case, and if the first two operations above correspond to the introduction of new values, that is, if $B \cup \beta \neq B$, alterations must correspond to new values. If all the A_i in F are disjoint, elements formed by these operations will also be disjoint. However, the third operation may give rise to intersections, but these may be eliminated by adding one element $(\alpha \setminus \bigcup (\text{dom } F)) \mapsto \beta$ to the function as a whole, rather than carrying out the operation for each element.

As an example of this process, consider the following sequence of patterns:

(red, y)
 (green, blue)
 (x, red)
 (green, green)
 (blue, y)

These patterns match the following pairs of values:

red \mapsto colour
 green \mapsto blue
 colour \mapsto red
 green \mapsto green
 blue \mapsto colour

Uniting the pairs according to these rules gives successively:

$$\begin{aligned}
&\{red \mapsto colour, green \mapsto blue\} \\
&\{red \mapsto colour, green \mapsto \{red, blue\}, blue \mapsto red\} \\
&\{\{red, green\} \mapsto colour, blue \mapsto red\} \\
&colour \mapsto colour
\end{aligned}$$

In formalising this process it is necessary to specify operations representing the difference and intersection of sets of values, just as it is already required to form unions. For differences and intersections, it is necessary to distinguish the empty set, which corresponds to the *incomplete* cover. The difference and intersection functions are both defined recursively and their types are given by:

$$difference, intersection : (Cover \times Cover) \mapsto Cover$$

Difference and intersection of tuples involves sets of pairs of differences, so it useful to define some schemas to provide the necessary signatures as follows:

| <i>Pair_element</i> | <i>Pair_set</i> |
|---|---|
| $A_1, A_2, \alpha_1, \alpha_2 : Cover$ | $F, result_pairs : \mathbb{P}(Cover \times Cover)$ |
| $res_pairs : \mathbb{P}(Cover \times Cover)$ | $\alpha_1, \alpha_2 : Cover$ |

In these schemas, $A_1 \mapsto A_2 \in F$ will be transformed into *res_pairs* as a result of adding a pair coverage element $\alpha_1 \mapsto \alpha_2$. Applying this process to all elements of *F* gives a new set, called *result_pairs*.

The intersection function is the simplest: for one element of a pair set the intersection is given by:

| <i>Int_element</i> |
|------------------------------------|
| <i>Pair_element</i> |
| $res_pairs = \{x \mapsto y\}$ |
| where |
| $x == intersection(A_1, \alpha_1)$ |
| $y == intersection(A_2, \alpha_2)$ |

The set of pairs is obtained by adding together all the elements and discarding any that are empty:

| | |
|--|--|
| <i>Int_pair</i> | |
| <i>Pair_set</i> | |
| $ \begin{aligned} \text{result_pairs} = \{ & \text{Int_element}; x, y : \text{Cover} \\ & A_1 \mapsto A_2 \in F \\ & \wedge x \mapsto y \in \text{res_pairs} \wedge x \neq \text{incomplete} \wedge y \neq \text{incomplete} \\ & \bullet x \mapsto y \\ & \} \end{aligned} $ | |

With this, the definition of *intersection* can be given by cases on the structure of *Cover* as follows:

| | |
|--|--|
| <i>IPrimitive</i> | |
| $c_1, c_2, c : \text{Cover}$ | |
| $ \begin{aligned} & (c_1 = \text{complete} \wedge c = c_2) \vee (c_1 = \text{incomplete} \wedge c = \text{incomplete}) \\ & \vee \\ & (c_2 = \text{complete} \wedge c = c_1) \vee (c_2 = \text{incomplete} \wedge c = \text{incomplete}) \end{aligned} $ | |

A complete cover accounts for all values and so the result is unchanged. An incomplete cover corresponds to the empty set which cannot have an intersection with any set of values.

For constructed covers, the intersection is determined by whether the constructors are equal and if so, whether the parameter values intersect:

| | |
|---|--|
| <i>IConstructions</i> | |
| $c_1, c_2, c : \text{Cover}$ | |
| $ \begin{aligned} & c_1 \in \text{ran construct} \wedge c_2 \in \text{ran construct} \\ & F' = \{ \} \wedge c = \text{incomplete} \\ & \vee \\ & F' \neq \{ \} \wedge c = \text{construct } F' \\ & \text{where} \\ & F == \text{construct}^{-1} c_1 \\ & F' == \{ c : \text{dom } F; \text{ parc, int} : \text{Cover} \\ & \quad c \mapsto \text{parc} \in \text{construct}^{-1} c_2 \\ & \quad \wedge \text{int} = \text{intersection}(F \ c, \text{parc}) \neq \text{incomplete} \\ & \quad \bullet c \mapsto \text{int} \\ & \} \end{aligned} $ | |

For tuples, any member of the *pair* coverages may intersect:

| |
|---|
| <i>ITuples</i> |
| $c_1, c_2, c : Cover$ |
| $c_1 \in ran\ pair \wedge c_2 \in ran\ pair$ $F' = \{\} \wedge c = incomplete$ \vee $F' \neq \{\} \wedge c = pair\ F'$ <i>where</i> $F == pair^{-1}\ c_1$ $F' == \bigcup \{Int_pair \mid \alpha_1 \mapsto \alpha_2 \in pair^{-1}\ c_2 \bullet result_pairs\}$ |

Finally, the intersection function is given by the constraint:

$intersection = \lambda\ c_1, c_2 : Cover$
 $\bullet\ \mu\ c : Cover$
 $\mid\ IPrimitive \vee IConstructions \vee ITuples$
 $\bullet\ c$

The difference function is similar, except that for tuples, the difference of cartesian product sets is slightly more complicated. On one element of a pair set, the difference function carries out the following operation:

| |
|---|
| <i>Diff_element</i> |
| <i>Pair_element</i> |
| $res_pairs = \{x \mapsto A_2, z \mapsto y\}$ <i>where</i> $x == difference(A_1, \alpha_1)$ $y == difference(A_2, \alpha_2)$ $z == intersection(A_1, \alpha_1)$ |

For the pair relation:

| |
|--|
| <i>Diff_pair</i> |
| <i>Pair_set</i> |
| $result_pairs = \{Diff_element; x, y : Cover$ $\mid\ A_1 \mapsto A_2 \in F$ $\wedge\ x \mapsto y \in res_pairs \wedge x \neq incomplete \wedge y \neq incomplete$ $\bullet\ x \mapsto y$ $\}$ |

The definition of *difference* is now given by cases on the structure of *Cover* as follows:

DPrimitive

$c_1, c_2, c : \text{Cover}$

$c_1 = \text{incomplete} \wedge c = \text{incomplete}$

$c_2 = \text{complete} \wedge c = \text{incomplete} \vee c_2 = \text{incomplete} \wedge c = c_1$

The constraints express the fact that subtracting from the empty set must leave the empty set as should subtracting the complete set of values. Subtracting the empty set must leave the result unchanged.

For constructors, the new constructor coverage is obtained from the old one by modifying the coverage function where it has elements in common with the subtrahend:

DConstructions

$c_1, c_2, c : \text{Cover}$

$c_1 \in \text{ran construct} \wedge c_2 \in \text{ran construct}$

$F' = \{\} \wedge c = \text{incomplete}$

\vee

$F' \neq \{\} \wedge c = \text{construct } F'$

where

$F == \text{construct}^{-1} c_1$

$F' == \{c : \text{dom } F; \text{ parc, diff} : \text{Cover}$

$| c \mapsto \text{parc} \in \text{construct}^{-1} c_2$

$\wedge \text{diff} = \text{difference}(F c, \text{parc}) \neq \text{incomplete}$

\vee

$c \notin \text{dom}(\text{construct}^{-1} c_2) \wedge \text{diff} = F c$

$\bullet c \mapsto \text{diff}$

$\}$

When both coverages are pairs the difference is given by:

DTuples

$c_1, c_2, c : \text{Cover}$

$c_1 \in \text{ran pair} \wedge c_2 \in \text{ran pair}$

$F' = \{\} \wedge c = \text{incomplete}$

\vee

$F' \neq \{\} \wedge c = \text{pair } F'$

where

$F == \text{pair}^{-1} c_1$

$F' == \bigcup \{\text{Diff_pair} \mid \alpha_1 \mapsto \alpha_2 \in \text{pair}^{-1} c_2 \bullet \text{result_pairs}\}$

When c_1 is complete, but c_2 is not, it is necessary to expand c_1 into the appropriate representation of completeness. For constructors, we have the following:

$$\begin{array}{l}
 \text{cl_complete_c2_constructor} \text{ —————} \\
 c_1, c_2, c : \text{Cover} \\
 \hline
 c_1 = \text{complete} \wedge c_2 \in \text{ran construct} \\
 F' = \{\} \wedge c = \text{incomplete} \\
 \vee \\
 F' \neq \{\} \wedge c = \text{construct } F' \\
 \text{where} \\
 \text{conset} == \mu cs : \text{ran datatype} \mid \text{dom}(\text{construct}^{-1} c_2) \subseteq cs \bullet cs \\
 F' == \{c : \text{conset}; \text{parc}, \text{diff} : \text{Cover} \\
 \quad \mid c \mapsto \text{parc} \in \text{construct}^{-1} c_2 \\
 \quad \wedge \text{diff} = \text{difference}(\text{complete}, \text{parc}) \neq \text{incomplete} \\
 \quad \bullet c \mapsto \text{diff} \\
 \quad \}
 \end{array}$$

and for tuples:

$$\begin{array}{l}
 \text{cl_complete_c2_pair} \text{ —————} \\
 c_1, c_2, c : \text{Cover} \\
 \hline
 c_1 = \text{complete} \wedge c_2 \in \text{ran pair} \\
 F' = \{\} \wedge c = \text{incomplete} \\
 \vee \\
 F' \neq \{\} \wedge c = \text{pair } F' \\
 \text{where} \\
 F == \{\text{complete} \mapsto \text{complete}\} \\
 F' == \bigcup \{\text{Diff_pair} \mid \alpha_1 \mapsto \alpha_2 \in \text{pair}^{-1} c_2 \bullet \text{result_pairs}\}
 \end{array}$$

This gives for the difference function:

$$\begin{array}{l}
 \text{difference} = \lambda c_1, c_2 : \text{Cover} \\
 \quad \bullet \mu c : \text{Cover} \\
 \quad \quad \mid \text{DPrimitive} \vee \text{DConstructions} \vee \text{DTuples} \\
 \quad \quad \vee \text{cl_complete_c2_constructor} \vee \text{cl_complete_c2_pair} \\
 \quad \bullet c
 \end{array}$$

The union function is similar in structure to difference, but this time it is also necessary to take into account the unique representation of completeness. (The intersection and difference functions cannot generate a *complete* cover, although they may generate an *incomplete* one. For the *union* function, the converse is true.) To

express completeness, it is necessary to define a relation *UNION*, corresponding to a distributed union function:

$$\begin{array}{l}
 U : \text{seq Cover} \rightarrow \text{Cover} \\
 U = \lambda sc : \text{seq Cover} \\
 \quad \bullet \mu c : \text{Cover} \\
 \quad \quad | sc = \langle \rangle \wedge c = \text{incomplete} \\
 \quad \quad \vee \\
 \quad \quad sc \neq \langle \rangle \wedge c = \text{union}(\text{hd } sc, U(\text{tl } sc)) \\
 \quad \bullet c
 \end{array}$$

$$\begin{array}{l}
 \text{_UNION_} == \{cs : \mathbb{P} \text{ Cover}; c : \text{Cover} \\
 \quad | \exists sc : \text{seq Cover} \\
 \quad \quad | \text{ran } sc = cs \wedge \#sc = \#cs \\
 \quad \quad \bullet c = U \text{ } sc \\
 \quad \quad \bullet c \mapsto cs \\
 \quad \}
 \end{array}$$

This is rather unsatisfactory as *UNION* has had to be defined as a relation, rather than a function. This is because the definition depends on the order in which the covers are united, specified by the sequence of covers, *sc*, above. For type-compatible covers, the result should be independent of the order, corresponding to the fact that set union distributes.

With the *UNION* relation, the completeness of a set of tuples may be expressed as follows:

$$\begin{array}{l}
 \text{Union_complete} \\
 c : \text{Cover}; F' : \mathbb{P} (\text{Cover} \times \text{Cover}) \\
 \quad cc = \text{complete} \wedge c = \text{complete} \\
 \quad \vee \\
 \quad cc = \text{incomplete} \wedge c = \text{pair } F' \\
 \quad \vee \\
 \quad cc \neq \text{incomplete} \wedge cc \neq \text{complete} \\
 \quad \wedge c = \text{pair}(\text{completed} \triangleleft F' \cup \{cc \mapsto \text{complete}\}) \\
 \text{where} \\
 \quad \text{completed} : \mathbb{P} \text{ Cover}; cc : \text{Cover} \\
 \quad \text{completed} = \{c : \text{dom } F' \mid F' c = \text{complete}\} \\
 \quad cc \text{ UNION completed}
 \end{array}$$

In this schema, if any of the second elements is complete, then the corresponding

first elements may be united. If the result is complete then the set of pairs is also complete.

The union function can now be defined in an analogous way to difference:

| | |
|---|-------|
| <i>Union_element</i> | _____ |
| <i>Pair_element</i> | _____ |
| <hr/> | |
| $z = A_2 \wedge res_pairs = \{A_1 \mapsto A_2\}$ | |
| \vee | |
| $z \neq A_2 \wedge res_pairs = \{x \mapsto A_2, y \mapsto z\}$ | |
| where | |
| $x == difference(A_1, \alpha_1)$ | |
| $y == intersection(A_1, \alpha_1)$ | |
| $z == union(\alpha_2, A_2)$ | |

For the pair relation, a constraint to express the third difference operation for the set of pairs as a whole must be added:

| | |
|---|-------|
| <i>Union_pair</i> | _____ |
| <i>Pair_set</i> | _____ |
| <hr/> | |
| $remainder = incomplete \wedge result_pairs = pairs1$ | |
| \vee | |
| $remainder \neq incomplete \wedge result_pairs = pairs1 \cup \{remainder \mapsto \alpha_2\}$ | |
| where | |
| $pairs1 : \mathbb{P} (Cover \times Cover); \text{ cf, } remainder : Cover$ | |
| <hr/> | |
| $pairs1 = \{ Union_element; x, y : Cover$ | |
| $\mid A_1 \mapsto A_2 \in F$ | |
| $\wedge x \mapsto y \in res_pairs \wedge x \neq incomplete \wedge y \neq incomplete$ | |
| $\bullet x \mapsto y$ | |
| $\}$ | |
| cf UNION dom F | |
| $remainder = difference(\alpha_1, cf)$ | |

With these schemas, forming the union of tuples is given by:

| |
|---|
| <i>UTuples</i> |
| $c_1, c_2, c : \text{Cover}$ |
| $c_1 \in \text{ran pair} \wedge c_2 \in \text{ran pair}$ <i>Union_complete</i> <i>where</i> $F == \text{pair}^{-1} c_1$ $F' == \bigcup \{ \text{Union_pair} \mid \alpha_1 \mapsto \alpha_2 \in \text{pair}^{-1} c_2 \bullet \text{result_pairs} \}$ |

The primitive coverages are easily dealt with. If the coverage is already *complete* adding more values does not increase it; if the *complete* coverage is added, the result can only be *complete*; if the *incomplete* coverage is added, the set of values covered is unchanged:

| |
|---|
| <i>UPrimitive</i> |
| $c_1, c_2, c : \text{Cover}$ |
| $(c_1 = \text{complete} \vee c_2 = \text{complete}) \wedge c = \text{complete}$ \vee $(c_1 = \text{incomplete} \wedge c = c_2) \vee (c_2 = \text{incomplete} \wedge c = c_1)$ |

And now the union function is given by

```

union =  $\lambda c_1, c_2 : \text{Cover}$ 
      •  $\mu c : \text{Cover}$ 
      | UPrimitive  $\vee$  UConstructions  $\vee$  UTuples
      •  $c$ 

```

Z_match_funs keeps *CPrimitive*, *CConstructions*, *CTuples*,
UPrimitive, *UConstructions*, *UTuples*,
intersection, *IPrimitive*, *IConstructions*, *ITuples*,
difference, *DPrimitive*, *DConstructions*, *DTuples*

6 Proof opportunities

Z_match_spec : *Module*

Z_match_AI : *Module*

Z_match_ops : *Module*

Z_match_funs : *Module*

The section title is intended to convey the concept of selective proof: proof should be used to increase confidence in those parts of the design which need further investigation, rather than calling for the proof of everything from first principles. In a specification of this nature, there are two sorts of proof opportunity, namely those associated with the consistency of the specification and those associated with the refinement process itself. In the course of developing this implementation, the opportunity has been taken to point out various consistency proofs as the need arises. For example, datatypes should be satisfiable, μ -terms should stand for uniquely existing values and the use of function arrows should be compatible with the axiomatic definition of the functions. The proof requirements for consistency are relatively trivial and do not add greatly to the understanding of the problem. Consequently we shall concentrate on the proof opportunities generated by the refinement itself and attempt to show the main structure of the proofs. Even here, the sea of theorems to prove is both wide and deep, so we shall concentrate on one or two example cases.

6.1 Proofs for the *coverage* function

The main theorem to prove is

Pattern_Compatible

\vdash

Abs_fn(coverage(p), type) = valcover(p, type)

The theorem contains *p* and *type* as a parameter, so it has to be shown for all values of these types. The constraint in the hypothesis ensures that the *type* is determined by the pattern in most cases, so the main proof is by induction over the structure of *Pattern*, with *type* corresponding.

Establishing a theorem of the form $\mathcal{P}(p)$, where *p* is a *Pattern* and \mathcal{P} some predicate on *p*, requires establishing the following base cases:

$\vdash \mathcal{P}(\text{unit } p)$

sc : *SCONST* $\vdash \mathcal{P}(\text{sconst } p(\text{sc}))$

v : *Variable* $\vdash \mathcal{P}(\text{var } p(v))$

and the following induction steps

$$\text{Cons_patt}; \mathcal{P}(\text{patt}) \vdash \mathcal{P}(\text{consp}(\theta \text{Cons_patt}))$$

$$\text{tp} : \text{tuple Pattern}; \forall p : \text{ran tp} \bullet \mathcal{P}(p) \vdash \mathcal{P}(\text{tupp}(\text{tp}))$$

For the coverage theorem, the base cases are a consequence of the following theorems whose proof is immediate:

$$\text{PCPrimitive}; c : \text{Cover}; \text{vals} : \mathbb{P} \text{Value}; p = \text{unitp}$$

$$\vdash$$

$$\text{AIPrimitive} \wedge \text{CPrimitive} \wedge \text{vals} = \{v : \text{type} \mid \text{MPrimitive}\}$$

$$\text{PCPrimitive}; c : \text{Cover}; \text{vals} : \mathbb{P} \text{Value}; p \in \text{ran sconstp}$$

$$\vdash$$

$$\text{AIPrimitive} \wedge \text{CPrimitive} \wedge \text{vals} = \{v : \text{type} \mid \text{MPrimitive}\}$$

$$\text{PCPrimitive}; c : \text{Cover}; \text{vals} : \mathbb{P} \text{Value}; p \in \text{ran var}$$

$$\vdash$$

$$\text{AIPrimitive} \wedge \text{CPrimitive} \wedge \text{vals} = \{v : \text{type} \mid \text{MPrimitive}\}$$

For the induction steps, the coverage property required may be expressed using the following schema:

| |
|--|
| $\text{Coverage_property} \text{ ————— }$ $p : \text{Pattern}; \text{type} : \text{Type}$ |
| $\text{Pattern_Compatible} \Rightarrow$ $\text{Abs_fn}(\text{coverage}(p), \text{type}) = \text{valcover}(p, \text{type})$ |

The constructor case will then follow from the theorem:

$$p : \text{Pattern}; \text{type} : \text{Type}; c : \text{Cover}; \text{vals} : \mathbb{P} \text{Value}$$

$$\text{Cons_patt}; p = \text{consp } \theta \text{Cons_patt}$$

$$\text{PCConstrutions}; \text{partype} : \text{Type}; \text{partype} = \text{type_of_con con}$$

$$\text{Coverage_property}_{[patt/p, \text{partype/type}]}$$

$$\vdash$$

$$\text{AIConstrutions} \wedge \text{CConstrutions} \wedge \text{vals} = \{v : \text{type} \mid \text{MConstrutions}\}$$

while the tuple case follows from the theorem:

$p : \text{Pattern}; \text{type} : \text{Type}; c : \text{Cover}; \text{vals} : \mathbb{P} \text{ Value}$
 $tp : \text{tuple Pattern}; p = \text{tupp } tp$
 $PCTuples$
 $\pi : \text{seq Type}; \pi = \{i : \text{dom } tp \bullet i \mapsto \{tv : \text{tupv}^{-1} \llbracket \text{type} \rrbracket \bullet tv \ i\}\}$
 $\forall i : \text{dom } tp; p : \text{ran } tp; \text{type} : \text{ran } \pi$
 $| p = tp \ i \wedge \text{type} = \pi \ i$
 $\bullet \text{Coverage_property}$
 \vdash
 $ALTuples \wedge CTuples \wedge \text{vals} = \{v : \text{type} \mid MTuples\}$

In both of these theorems, the hypothesis list makes use of the fact that the type in the induction hypothesis must be derived from the given type, which is compatible with the pattern. This is expressed by the following theorems:

$p : \text{Pattern}; \text{type} : \text{Type}$
 $\text{Cons_patt}; p = \text{consp } \theta \text{Cons_patt}$
 $\text{Pattern_Compatible}_{[patt/p, partype/type]}$
 \vdash
 $partype = \text{type_of_con } con$

$p : \text{Pattern}; \text{type} : \text{Type}$
 $tp : \text{tuple Pattern}; p = \text{tupp } tp$
 $\pi : \text{seq Type}$
 $\forall i : \text{dom } tp; p : \text{ran } tp; \text{type} : \text{ran } \pi$
 $| p = tp \ i \wedge \text{type} = \pi \ i$
 $\bullet \text{Pattern_Compatible}$
 \vdash
 $\pi = \{i : \text{dom } tp \bullet i \mapsto \{tv : \text{tupv}^{-1} \llbracket \text{type} \rrbracket \bullet tv \ i\}\}$

The proof of the constructor case may be carried out along the following lines. The conclusion consists of a conjunction of three predicates, each of which must be shown to be true. The second provides a value for the cover, c , which may be used to rewrite the first predicate to give

$p : \text{Pattern}; \text{type} : \text{Type}; c : \text{Cover}; \text{vals} : \mathbb{P} \text{ Value}$
 $\text{Cons_patt}; p = \text{consp } \theta \text{Cons_patt}$
 $partype : \text{Type}; partype = \text{type_of_con } con$
 $PCConstructions; \text{Coverage_property}_{[patt/p, partype/type]}$
 \vdash
 $\text{vals} = \{\text{val} : \text{Abs_fn}(\text{coverage } patt, \text{type_of_con } con) \bullet \text{consv } \theta \text{Cons_val}\}$
 $\text{vals} = \{\text{val} : \text{type_of_con } con \mid patt \text{ matches } val \bullet \text{consv } \theta \text{Cons_val}\}$

This theorem follows immediately from the hypothesis. The tuple case requires an

induction over the length of the tuple, starting with the base case of the tuple size being 2. The constructor and tuple case together give the theorem required.

The other requirements on *coverage* are easily established. As an example, we can try and establish

$$\begin{array}{l} \text{Pattern_Compatible}; c : \text{Cover}; \text{vals} : \mathbb{P} \text{Value} \\ c = \text{coverage}(p) \wedge \text{vals} = \text{valcover}(p, \text{type}) \\ \vdash \\ c = \text{incomplete} \vee \text{vals} \neq \{\} \end{array}$$

The base cases are a consequence of the following theorems whose proof is immediate.

$$\begin{array}{l} \text{PCPrimitive}; \text{CPrimitive}; \text{vals} : \mathbb{P} \text{Value} \\ p = \text{unitp} \wedge \text{vals} = \{v : \text{type} \mid \text{MPrimitive}\} \\ \vdash \\ c \neq \text{incomplete} \wedge \text{vals} = \{\text{unirv}\} \end{array}$$

$$\begin{array}{l} \text{PCPrimitive}; \text{CPrimitive}; \text{vals} : \mathbb{P} \text{Value} \\ p \in \text{ran sconstp} \wedge \text{vals} = \{v : \text{type} \mid \text{MPrimitive}\} \\ \vdash \\ c = \text{incomplete} \end{array}$$

$$\begin{array}{l} \text{PCPrimitive}; \text{CPrimitive}; \text{vals} : \mathbb{P} \text{Value} \\ p \in \text{ran var} \wedge \text{vals} = \{v : \text{type} \mid \text{MPrimitive}\} \\ \vdash \\ c \neq \text{incomplete} \wedge \text{vals} = \text{type} \end{array}$$

The induction property is

| |
|---|
| $\begin{array}{l} \text{Incomplete_property} \\ p : \text{Pattern}; \text{type} : \text{Type} \end{array}$ |
| $\begin{array}{l} \text{Pattern_Compatible} \Rightarrow c \neq \text{incomplete} \Rightarrow \text{vals} \neq \{\} \\ \text{where} \\ c == \text{coverage}(p) \\ \text{vals} == \text{valcover}(p, \text{type}) \end{array}$ |

The constructor induction step is as follows

$p : \text{Pattern}; \text{type} : \text{Type}; c : \text{Cover}; \text{vals} : \mathbb{P} \text{ Value}$
 $\text{Cons_patt}; p = \text{consp } \theta \text{Cons_patt}$
 $\text{PCConstructions}; \text{partype} : \text{Type}; \text{partype} = \text{type_of_con } \text{con}$
 $\text{Coverage_property}_{\{\text{patt}/p, \text{partype}/\text{type}\}}$
 \vdash
 $\text{CConstructions} \wedge \text{vals} = \{v : \text{type} \mid \text{MConstructions}\}$
 $c \neq \text{incomplete} \Rightarrow \text{coverage } \text{patt} \neq \text{incomplete}$

From the conclusion and the hypothesis it is possible to show that $\text{valcover}(\text{patt}, \text{type_of_con } \text{con})$ is not empty, from which it follows that vals is not empty. As before, the tuple case will involve an induction over the size of the tuple.

To summarise, the proofs of 4 theorems are required, as listed in section 5, of which the first two have been outlined here. The third theorem follows immediately from the definition of Abs_fn while the fourth will have a similar structure to that given above. Thus to show the main structure of the proof fully we are required to display 18 theorems: 5 each for the non-trivial theorems corresponding to the constructors of Pattern , 2 for the type lemmas and one for the third, trivial, theorem.

6.2 Proofs for the *union* function

The main theorem we have to show is

Union
 \vdash
 $\text{Abs_fn}(c, \text{type}) = \text{Abs_fn}(c_1, \text{type}) \cup \text{Abs_fn}(c_2, \text{type})$

Two lemmas are needed for this, which prove that the difference and intersection functions have their intended effect. The intersection lemma is the easiest of these, and may be expressed using the schema below:

| |
|---|
| Intersection $c_1, c_2, c : \text{Cover}$ $\text{type} : \text{Type}$ |
| $c_1 \text{ cover_compatible } \text{type} \wedge c_2 \text{ cover_compatible } \text{type}$ $c = \text{intersection}(c_1, c_2)$ |

The theorem required is

Intersection
 \vdash
 $\text{Abs_fn}(c, \text{type}) = \text{Abs_fn}(c_1, \text{type}) \cap \text{Abs_fn}(c_2, \text{type})$

The proof involves a structural induction over a cross-product of Cover , again with type

corresponding. There are 4 constructors in the datatype and consequently 16 cases to prove. Of these, the two cases with mixed *pair* and *construct* constructors may be eliminated because they cannot be simultaneously compatible with the type. Of the 14 remaining cases, 12, which deal with the c_1 or c_2 being *complete* or *incomplete*, are trivial and follow immediately from the fact that the abstraction function delivers the full set of values in the type, or the empty set, respectively. As a sample theorem required for the full proof, take the case of a pair of *construct* coverages. The necessary theorem will be written in terms of the following induction property:

$$\begin{array}{|l}
 \text{Intersect_property} \\
 \hline
 c_1, c_2, c : \text{Cover}; \text{type} : \text{Type} \\
 \hline
 \text{Intersection} \Rightarrow \text{vals} = \text{vals}_1 \cap \text{vals}_2 \\
 \text{where} \\
 \text{vals}_1 == \text{Abs_fn}(c_1, \text{type}) \\
 \text{vals}_2 == \text{Abs_fn}(c_2, \text{type}) \\
 \text{vals} == \text{Abs_fn}(c, \text{type})
 \end{array}$$

With this, the constructor induction step is

$$\begin{array}{l}
 c_1, c_2, c : \text{Cover}; \text{vals}_1, \text{vals}_2, \text{vals} : \# \text{Value}; \text{type} : \text{Type} \\
 F_1, F_2 : \text{CON} \Rightarrow \text{Cover}; c_1 = \text{construct } F_1; c_2 = \text{construct } F_2 \\
 A1_1; A1_2; A1; \text{type}_1 = \text{type}_2 = \text{type} \\
 \text{CCConstructions}_1; \text{CCConstructions}_2 \\
 \forall \text{con} : \text{dom } F_1 \cup \text{dom } F_2; \text{type} : \text{Type}; c_1, c_2, c : \text{Cover} \\
 | \text{type} = \text{type_of_con } \text{con} \wedge c_1 = F_1 \text{ con} \wedge c_2 = F_2 \text{ con} \\
 \bullet \text{Intersect_property} \\
 \vdash \\
 \text{IConstructions} \Rightarrow \text{vals} = \text{vals}_1 \cap \text{vals}_2
 \end{array}$$

This impressive list of hypotheses is generated quite mechanically. The theorem may be proved by showing $\forall v : \text{vals} \bullet v \in \text{vals}_1 \wedge v \in \text{vals}_2$. This is straightforward, but rather tedious, and complicated by the extra test on incompleteness.

As a final sample, the proof that an increment to the coverage implies an increment to the values covered will be exhibited. The theorem to prove is

$$\begin{array}{l}
 \text{Union} \\
 \vdash \\
 c \neq c_2 \Rightarrow \text{Abs_fn}(c, \text{type}) \neq \text{Abs_fn}(c_2, \text{type})
 \end{array}$$

The base cases are relatively easily established, so we shall consider the induction steps. As before define a schema to express the induction hypothesis:

| |
|--|
| <i>Increment_property</i> $c_1, c_2, c : \text{Cover}; \text{type} : \text{Type}$ |
| $\text{Union} \Rightarrow c \neq c_2 \Rightarrow \text{vals} \neq \text{vals}_2$ <i>where</i> $\text{vals} == \text{Abs_fn}(c, \text{type})$ $\text{vals}_2 == \text{Abs_fn}(c_2, \text{type})$ |

Taking the tuple induction step this time, the theorem to be proved is as follows:

$$\begin{array}{l}
c_1, c_2, c : \text{Cover}; \text{vals}_1, \text{vals}_2, \text{vals} : \mathbb{P} \text{Value}; \text{type} : \text{Type} \\
F_1, F_2 : \mathbb{F} (\text{Cover} \times \text{Cover}); c_1 = \text{pair } F_1; c_2 = \text{pair } F_2 \\
\text{AI}_1; \text{AI}_2; \text{AI}; \text{type}_1 = \text{type}_2 = \text{type} \\
\text{CCTuples}_1; \text{CCTuples}_2 \\
\text{hdtype}, \text{iltype} : \text{Type}; \text{hdtype} = \text{HD type} \wedge \text{iltype} = \text{TL type} \\
\forall c_1 : \text{dom } F_1; c_2 : \text{dom } F_2; \text{type} : \text{Type}; c : \text{Cover} \\
| \text{type} = \text{hdtype} \\
\bullet \text{Increment_property} \\
\forall c_1 : \text{ran } F_1; c_2 : \text{ran } F_2; \text{type} : \text{Type}; c : \text{Cover} \\
| \text{type} = \text{iltype} \\
\bullet \text{Increment_property} \\
\vdash \\
\text{UTuples} \Rightarrow c \neq c_2 \Rightarrow \text{vals} \neq \text{vals}_2
\end{array}$$

This theorem may be established by showing that $\exists v : \text{vals}_1 \bullet v \notin \text{vals}_2$. As with all the tuple cases, this is not particularly easy to prove although the existential witness is easy to find.

To summarise for the *union* case, five proofs are required, as listed in section 5.2, of which the second and fourth are immediate. The first proof breaks down into the *intersection*, *difference* and *union* cases, each based on a cross product of *Cover* each requiring the proof of 16 subsidiary cases, 48 in all. There will also be 2 subsidiary lemmas for manipulating the type in the induction property, just as for *coverage*. The third and fifth theorems also give rise to 16 cases so there are 82 subsidiary goals to establish. Of these the tuple cases are relatively complicated and would need to be broken down into further goals.

7 The implementation

This section gives the text of the Algol68 module which implements the design specified. It is included to give some idea of the formal distance between the design specification derived in section 5 and an actual implementation language. The implementation is part of an ML interpreter and so covers the aspects which were simplified in the design specification (apart from the exact treatment of the special constants). The main structure of the implementation follows fairly closely that of the specification inasmuch as it is easy to relate the parts of the implementation to the parts of the specification. The Z datatypes *Pattern* and *Cover* are implemented by the Algol68 *PATTERN* and *COVER* respectively and the *coverage* and *union* functions have the same name. The checking operation, *Check_op_2*, is implemented by the Algol68 *addpatt* and the test in the final operation is given by the procedure *comp*. However, the formal distance, in the sense of the theorems necessary to demonstrate the refinement, is still large.

In particular, further steps of data refinement have been undertaken. The constructor coverage measure, which in Z has been represented by $CON \Rightarrow Cover$, has been data refined into $N \Rightarrow Cover$, which is implemented as the Algol68 array of *cover*, *[]COVER*. In addition, the implementation uses a *total* function from the set of constructors in the datatype, rather than a partial function whose domain is the set of constructors already encountered. In the implementation, the constructors which have not been encountered are given an *incomplete* coverage.

Similarly, for tuples, the Z representation of *pair* $\ll \mathbb{F} (Cover \times Cover) \gg$ is implemented by the Algol68 mode *PAIR* = *STRUCT*(*COVER* *a*, *b*, *REF PAIR next*). In this case the abstract sets are being implemented by linked lists, with a natural implementation of universal quantifiers in terms of loops.

In the *union* function, there has been some operation refinement of the test for completion, the calculation of the remainder coverage and the test for whether the coverage has changed. These are all done during the execution of the main loop in the function, rather than sequentially as would result from a simple-minded implementation of the specification. As a result the *union* function delivers both the new coverage and an indication of whether it has changed.

ml_testmatch:

```
ml_modes :Module
ml_runtime :Module
ml_comp_mode :Module
```

```
MODE COVER,
  CONSTRUCT = STRUCT(REF[]COVER cvs),
  PAIR = STRUCT(COVER a, b, REF PAIR next),
  COVER = UNION(BOOL, {TRUE = complete, FALSE = incomplete}
    REF CONSTRUCT,
    REF PAIR
```

```

);

REF PAIR no_pair = NIL;

PROC construct = (REF[]COVER cvs)COVER:
    (HEAP CONSTRUCT c;  cvs OF c := cvs;  c);

PROC incomp = (COVER c)BOOL: CASE c IN (BOOL b): NOT b OUT FALSE ESAC;
{test for incomplete cover}

PROC comp = (COVER c)BOOL: CASE c IN (BOOL b): b OUT FALSE ESAC;
{test for complete cover}

PROC coverage = (PATTERN patt)COVER:
    CASE patt
    IN (VOID): TRUE {Unit constant is complete}
    , (BOOL b): {Boolean special constants}
        (HEAP[1:2]COVER valcover := (FALSE, FALSE);
         valcover[ABS b + 1] := TRUE;
         construct(valcover)
        )
    , (INT): FALSE    {integer indicates special constant which is incomplete}
    , (SHORT REAL): FALSE {...}
    , (LINE): FALSE {...}
    , (VAR): TRUE     {Variables are complete}
    , (REF CONSTANT c):
        (REF TYCON tycon = tycons[tyconno OF c];
         INT nocons = nocons OF tycon;
         IF nocons /= 1
         THEN HEAP[1:nocons]COVER valcover;
              FOR i TO nocons DO valcover[i] := FALSE OD;
              valcover[consno OF c] := TRUE;
              construct(valcover)
         ELSE {datatype contains only one value, and this must be it}
              TRUE
         FI
        )
    , (REF CONSPATT cp):
        (REF TYCON tycon = tycons[tyconno OF cp];
         INT nocons = nocons OF tycon;
         COVER parcover = coverage(par OF cp);
         IF incomp(parcover) {test incomplete}
         THEN FALSE
         ELIF nocons = 1
         ANDTH comp(parcover) {test complete}
         THEN TRUE
         ELSE HEAP[1:nocons]COVER valcover;
              FOR i TO nocons DO valcover[i] := FALSE OD;
              valcover[consno OF cp] := parcover;
              construct(valcover)
         FI
        )
    , (REF REFPATT rp):
        coverage(patt OF rp) {as there is only one ref constructor}

```

```

, (REF LAYERED l):
    coverage(patt OF l)    {the equivalent pattern}
, (REF LISTPATT lp):
    (HEAP[1:2]COVER valcover := (FALSE, FALSE);
    IF lp IS nolistpatt {test nil}
    THEN valcover[1] := TRUE
    ELSE {:: is a constructor of a 2-tuple: ('a * 'a list) -> 'a list}
        HEAP PAIR pair := (coverage(hd OF lp), coverage(tl OF lp), NIL);
        valcover[2] := pair
    FI;
    construct(valcover)
)
, (REF TUPLEPATT tp):
    (REF VECTOR[]PATTERN patts = patts OF tp;  INT size = UPB patts;
    COVER a = coverage(patts[1]),
    b = IF size = 2
        THEN coverage(patts[2])
        ELSE TUPLEPATT tltip;  patts OF tltip := patts[2:];
        coverage(tltip)
    FI;
    HEAP PAIR := (a, b, NIL)
)
ESAC;

PROC intersection = (COVER c1, c2)COVER:
CASE c2
IN (BOOL b): IF b THEN c1 ELSE FALSE FI
OUSE c1
IN (BOOL b): IF b THEN c2 ELSE FALSE FI
    {the case elements above implement IPrimitive,
    the next one implements IConstructions}
, (REF CONSTRUCT cs1):
    CASE c2
    IN (REF CONSTRUCT cs2):
        (BOOL incomplete := TRUE;
        []COVER f1 = cvs OF cs1, f2 = cvs OF cs2;
        INT size = UPB f1;
        HEAP[1:size]COVER f;
        FOR i TO size
        DO IF NOT incomp(f[i] := intersection(f1[i], f2[i]))
            THEN incomplete := FALSE
        FI
        OD;
        IF incomplete THEN FALSE ELSE construct(f) FI
    )
    ESAC
, (REF PAIR p1): {This element implements ITuples}
    CASE c2
    IN (REF PAIR p2):
        (REF PAIR pp1, pp2 := p2, {pointers into the lists}
        fprime := NIL; {the result}
        WHILE pp2 ISNT no_pair {test not end of list}
        DO pp1 := p1;
        WHILE pp1 ISNT no_pair
        DO COVER a = intersection(a OF pp1, a OF pp2),

```

```

        b = intersection(b OF pp1, b OF pp2);
        IF NOT incomp(a)
        ANDTH NOT incomp(b)
        THEN fprime := HEAP PAIR := (a, b, fprime)
        FI;
        pp1 := next OF pp1
      OD;
      pp2 := next OF pp2
    OD;
    IF fprime IS no_pair
    THEN FALSE
    ELSE fprime
    FI
  )
ESAC
ESAC;

PROC difference = (COVER c1, c2)COVER:
  CASE c2
  IN (BOOL b): IF b THEN FALSE ELSE c1 FI
  OUSE c1
  IN (BOOL b):
    IF b
    THEN CASE c2
      IN (REF CONSTRUCT cs2):
        {c1 complete c2 constructor}
        (REF[])COVER f2 = cvs OF cs2;
        [1:UPB f2]COVER f1;
        FOR i TO UPB f1 DO f1[i] := TRUE OD;
        difference(construct(f1), c2)
      )
      , (REF PAIR p2):
        {c1 complete c2 tuple}
        (PAIR p1 := (TRUE, TRUE, NIL);
        difference(p1, c2)
        )
    )
    ESAC
  ELSE FALSE
  FI
  , (REF CONSTRUCT cs1):
    CASE c2
    IN (REF CONSTRUCT cs2):
      {DConstructions}
      (BOOL incomplete := TRUE;
      REF[]COVER f1 = cvs OF cs1, f2 = cvs OF cs2;
      INT size = UPB f1;
      HEAP[1:size]COVER f;
      FOR i TO size
      DO IF NOT incomp(f[i] := difference(f1[i], f2[i]))
        THEN incomplete := FALSE
        FI
      OD;
      IF incomplete THEN FALSE ELSE construct(f) FI
    )
    ESAC
  , (REF PAIR p1):
    CASE c2
    IN (REF PAIR p2):

```

```

        (REF PAIR pp1, pp2 := p2, {pointers into the lists}
          fprime := NIL; {the result}
        WHILE pp2 ISNT no_pair {test not end of list}
        DO pp1 := p1;
          WHILE pp1 ISNT no_pair
          DO COVER x = difference(a OF pp1, a OF pp2),
              y = difference(b OF pp1, b OF pp2),
              z = intersection(a OF pp1, a OF pp2);
            IF NOT incomp(x)
            THEN fprime := HEAP PAIR := (x, b OF pp1, fprime)
            FI;
            IF NOT incomp(y) ANDTH NOT incomp(z)
            THEN fprime := HEAP PAIR := (z, y, fprime)
            FI;
            pp1 := next OF pp1
          OD;
          pp2 := next OF pp2
        OD;
        IF fprime IS no_pair
        THEN FALSE
        ELSE fprime
        FI
      )
    ESAC
  ESAC;

MODE UNIONRES = STRUCT(COVER c, BOOL differs);
{This is what union will actual deliver. The boolean says whether c differs
from c2}

PROC union = (COVER c1, c2)UNIONRES:
  CASE c2
  IN (BOOL b):
    IF b
    THEN {c2 complete, result unchanged} (TRUE, FALSE)
    ELSE {c2 incomplete, result changed unless c1 incomplete}
      (c1, NOT incomp(c1))
    FI
  OUSE c1
  IN (BOOL b):
    IF b
    THEN {c1 complete, result unchanged unless c2 complete}
      (TRUE, NOT comp(c2))
    ELSE {c1 incomplete, result unchanged} (c2, TRUE)
    FI
  , (REF CONSTRUCT cs1):
    CASE c2
    IN (REF CONSTRUCT cs2):
      {UConstructions}
      (BOOL complete := TRUE, changed := FALSE;
        []COVER f1 = cvs OF cs1, f2 = cvs OF cs2;
        INT size = UPB f1;
        HEAP[1:size]COVER f;
        FOR i TO size
        DO UNIONRES res := union(f1[i], f2[i]);
          changed := changed OR differs OF res;
          IF NOT comp(f[i] := c OF res)

```

```

        THEN complete := FALSE
        FI
    OD;
    IF complete
    THEN {it must have changed} (TRUE, TRUE)
    ELIF changed
    THEN (construct(f), TRUE)
    ELSE (c2, FALSE)
    FI
)
ESAC
, (REF PAIR p1):
CASE c2
IN (REF PAIR p2):
    {UTuples}
    (REF PAIR pp1, pp2 := p2, {pointers into the lists}
        fprime := NIL; {the result}

    BOOL changed := FALSE;
    COVER completed := FALSE;
    PROC add = (COVER a, b)VOID:
        (changed := TRUE;
        IF comp(b)
        THEN completed := c OF union(a, completed)
        ELSE fprime := HEAP PAIR := (a, b, fprime)
        FI
        );

    WHILE pp2 ISNT no_pair {test not end of list}
    DO pp1 := p1;
    COVER remainder := a OF pp2;
    WHILE pp1 ISNT no_pair
    DO COVER x = difference(a OF pp1, a OF pp2),
        y = intersection(a OF pp1, a OF pp2);
    UNIONRES res = union(b OF pp1, b OF pp2);
    COVER z = c OF res;
    IF differs OF res
    THEN IF NOT incomp(x)
        THEN add(x, b OF pp1)
        FI;
        IF NOT incomp(y) ANDTH NOT incomp(z)
        THEN add(y, z)
        FI
    FI;
    remainder := difference(a OF pp2, a OF pp1);
    pp1 := next OF pp1
    OD;
    IF NOT incomp(remainder) THEN add(remainder, b OF pp2) FI;
    pp2 := next OF pp2
    OD;

    IF NOT changed
    THEN (c2, FALSE)
    ELIF comp(completed)
    THEN (TRUE, TRUE)
    ELSE IF NOT incomp(completed)
        THEN fprime := HEAP PAIR := (completed, TRUE, fprime)
        FI;
        (fprime, TRUE)

```

```

        FI
    )
    ESAC
ESAC;

PROC addpatt = (PATTERN p, COVER c2)COVER:
    {Equivalent to Check_op_2}
    (COVER c1 = coverage(p);
    UNIONRES res = union(c1, c2);
    IF NOT incomp(c1) ANDTH NOT differs OF res
    THEN warn("This pattern is redundant")
    FI;
    c OF res
    )

KEEP COVER, coverage, union, addpatt, comp
FINISH

```

8 Conclusions

Before discussing the implications of this case study, it is worth emphasising the fact that this report is about refinement, not about the precise form of the pattern matching algorithm to use in ML. Other studies, for example Baudinet and MacQueen [1987], Peyton Jones [1987], give algorithms for compiling pattern matching expressions, but the problem of demonstrating the correctness of the algorithm still remains. Given this approach to the formal development of software therefore, the questions arise as to whether it is actually helpful to the developers and whether it actually convinces the evaluators.

Taking the developers point of view first of all, when reading this case study, one motors along quite happily through sections 1 to 4 and then the going gets rough in section 5 and finally one gets bogged down in section 6. Section 5 is difficult because the problem is difficult. Although the intuitive idea of an exhaustive check is clear it is extremely hard to think of a general algorithm which convincingly copes with all cases. In addition, intuitively one thinks that the type structure must play a relatively minor role in the algorithm, but equally it must be present in the specification and the problem is to see what assumptions can be made about it. Given that, the difficulty in section 5 is quite understandable and this approach, namely the development of the abstraction invariant and its use in calculating the implementation required, is a sensible way of designing software. Note incidentally, that the "bottom-up" approach of designing the software and *then* attempting to show that it satisfies the specification is dangerous. Our work on this study was impeded by several months fruitlessly trying to prove that our intuitive algorithm was correct when it was not.

Given the investment in abstract design, represented by sections 1 to 5, the implementation becomes very easy. The implementation in section 7 was written, tested and debugged in 1 day. (Debugged? Yes, CTS wrote TRUE instead of FALSE and forgot to move on the list pointers in some of the loops. These faults will not be found by the formal method without *much* more formalism, but are relatively easily found by testing.) One can argue therefore that abstract algorithm design is cost effective as a production method, but this does not apply to the formal proof aspects detailed in section 6. This is because the actual development of the proof is *demanding*. And it is *demanding* because it is boring and it is boring because it does not lead to a much greater insight into the problem.

From the evaluator's point of view one can take a similar warm view to the process of abstract algorithm design exemplified by the first 5 sections, because these help the evaluator to understand the algorithm. Indeed, it is probably the only approach from which one can derive a convincing argument for correctness and consequently the only approach likely to satisfy an independent evaluation. It is however not clear the extent to which the proof process adds further assurance: at some points it is hard to see what is going on through all the detail of the formalism. Furthermore, it seems inevitable that the development will be informal at some stage, simply by reason of the number of proofs required. Section 6 covers only an outline of the major proof

steps so one could anticipate a proof document for what is a small part of an ML compiler being perhaps 10 times larger than this section. Moreover, this section is notable for what is omitted rather than what is included. At the higher level the formalisation of how the compiling functions are called has been omitted and at the lower level, there is at least as much operation refinement required to end up with the implementation language as the data refinement needed to end up with the design. The sheer number of proof steps generated makes it seem inconceivable that the pattern matching algorithm in an ML compiler will *ever* be carried out fully formally and mechanically checked.

It is not however clear that mechanical proof is always called for in the very highest levels of assurance. Using the DoD's evaluation criteria as an example [DoD 1985], one could equate the development recorded in sections 1 to 5 with the assurance level contained in the A1 criteria. The specification corresponds to the policy model; section 5 corresponds to the formal top level specification; the combination of formal and informal techniques showing the correspondence with the specification is contained in sections 3 to 5. The formalism has been mechanically checked for correct syntax and well-typing: the question is, is that enough to satisfy A1 assurance, or the equivalent UK confidence level 6, assured design.

Rather than answer this contentious question directly, we shall discuss two related ones, namely, whether the proof process of section 6 materially adds to the confidence in the software, and whether it is necessary to supply the missing formalism. Taking the latter question first, we do not believe that much would be gained by formalising the syntax analyser part of an ML compiler and relating it to the compiling operation specifications so that the pre-conditions for these operations were derived formally, rather than informally as has been done here. Syntax analysis is normally treated with a special purpose tool anyway and the pre-conditions are relatively easily checked by eye. The further refinement involved in arriving at the Algol68 implementation is more debateable. For the most part this is straightforward although one would like to see some code level verification of the loops in the *union* function. Note that the design specification is suggestive of the annotations which would be necessary for verification condition generation by tools such as MALPAS and Gypsy.

Whether the proof in general is adding assurance is a more subjective judgement. To give it an appropriate subjective context one could rephrase the question thus. Given that one is going to entrust one's life to a piece of software, which will be written to a fixed price by contractor A and evaluated to a fixed price by contractor B, what should A and B be required to do? Within this context the proof process is valuable because it gives an objective test for satisfaction. Mechanical aids are important in this, but their use should not cloud the understanding of the proof in the evaluator's mind. Mechanical proof should support the kind of informal proof common in mathematics, rather than supplanting it. In practice, the theorems contained in section 6 are at a level of detail which should be within the range of a mechanical theorem prover and at which an evaluator could well take the mechanical proof on trust.

Even with mechanical aids, it is unlikely that realistic problems will be capable of having the proof of all the theorems exhibited being carried out mechanically. For this particular case study, going down to this level of detail gives rise to at least a 100 theorems. Requiring the proof of all these theorems to be carried out mechanically for certification would be unreasonably costly. However a selective approach is sensible and enables the level of assurance to be related to the amount of effort spent in verification. For example, in this case study, the theorems are arranged in a tree: the refinement obligations are expressed by two theorems, which are broken down into 9 which expand into the 100 or so. A satisfactory technique would be to prove that the 9 theorems entailed the refinement obligations, thus demonstrating that all cases had been considered and then to take one of the branches of the proof tree, perhaps for the harder tuple case, down to the leaves of the tree. Even this might be too onerous and an evaluator could be satisfied with proving some, but not all, of the theorems on the way, simply in order to make the most cost-effective use of his time. A mechanical prover could help here by proving certain of the easier cases, so that these could simply be accepted by the evaluator.

To summarise:

- The formal development process, in particular the key step of exhibiting the abstraction invariant and using it to develop the abstract algorithm design, is a useful and cost-effective technique for developing software, quite apart from its use in *demonstrating correctness*.
- Formal proof adds to the assurance, but complete formality obscures. Proof needs to be undertaken within a context using both formal and informal elements. The structure of the proof and the way it is delivered to an evaluator (the proof document) ought to follow intuitive, informal, proof methods.

References

- DoD (1985). Department of Defense Trusted Computer System Evaluation Criteria, National Computer Security Center, Fort Meade, Maryland, USA.
- Baudinet M and MacQueen D (1987). Tree pattern matching for ML, in *Functional programming languages and computer architectures*, Gilles Kahn (ed), Lecture Notes in Computer Science 274, Springer Verlag.
- Gordon M J C, Milner R and Wadsworth C P (1979). Edinburgh LCF, Lecture Notes in Computer Science 78, Springer Verlag.
- Harper R, Milner R and Tofte M (1988). The definition of standard ML version 2. Laboratory for the Foundations of Computer Science, report ECS-LFCS-88-62, University of Edinburgh.
- Hayes I, (1987). Specification case studies, Prentice Hall International series in Computer Science, 1987.
- Jones, C B (1986). Systematic software development using VDM, Prentice-Hall.
- Morgan C C, Robinson K A (1987). Specification statements and refinement, IBM Journal of Research and Development, 31, 5.
- Morgan C C (1988). The specification statement, TOPLAS 10, 3.
- Paulson L C (1987). Logic and computation. Cambridge tracts in theoretical computer science 2, Cambridge University Press.
- Peyton Jones S L (1987). The implementation of functional programming languages, Prentice Hall International series in Computer Science.
- Spivey, J M (1988). Understanding Z: a specification language and its formal semantics, Cambridge University Press.
- Sufrin, B (1983). Formal system specification - notation and examples, in *Tools and Notations for Program Construction* (Neel ed.), Cambridge University Press.

DOCUMENT CONTROL SHEET

Overall security classification of sheet UNCLASSIFIED

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

| | | | | |
|---|--|------------------------------------|--|----------------|
| 1. DFIC Reference (if known) | 2. Originator's Reference Report 89004 | 3. Agency Reference | 4. Report Security Classification U/C | |
| 5. Originator's Code (if known) 7784000 | 6. Originator (Corporate Author) Name and Location ROYAL SIGNALS & RADAR ESTABLISHMENT ST ANDREWS ROAD, GREAT MALVERN WORCESTERSHIRE WR14 3PS | | | |
| 5a. Sponsoring Agency's Code (if known) | 6a. Sponsoring Agency (Contract Authority) Name and Location | | | |
| 7. Title Pattern matching in ML: a case study in refinement | | | | |
| 7a. Title in Foreign Language (in the case of translations) | | | | |
| 7b. Presented at (for conference papers) Title, place and date of conference | | | | |
| 8. Author 1 Surname, initials Macdonald L | 9(a) Author 2 Randell G P | 9(b) Authors 3,4... Sennett C T | 10. Date 1989.05 | pp. ref. 54 |
| 11. Contract Number | 12. Period | 13. Project | 14. Other Reference | |
| 15. Distribution statement UNLIMITED | | | | |
| Descriptors (or keywords) continue on separate piece of paper | | | | |
| Abstract This report is a case study in data refinement, that is the process of taking a formal specification written in terms of abstract values and converting it into a concrete form suitable for implementation. The case study takes a non-trivial problem, namely pattern matching in the language ML, and presents the refinement process and proof obligations incurred concluding with an implementation in Algol 68. The report concludes with a discussion of the strengths and weaknesses of the formal development process. | | | | |